REPORT DOCUMENTATION PAGE

AFRL-SR-AR-TR-05-

0420

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
15-08-2005	FINAL TECHNICAL REPORT	15-SEP-2005 – 15-JUL-2005

4. TITLE AND SUBTITLE
A FRAMEWORK FOR MODELING AND ANALYZING COMPLEX DISTRIBUTED SYSTEMS

5a. CONTRACT NUMBER
FA9550-04-C-0084

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
LYNCH, NANCY A
SHVARTSMAN, ALEXANDER A.

5d. PROJECT NUMBER
N/A

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
001AC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
VEROMODO, INC.
11 OSBORNE ROAD
BROOKLINE, MA 02446

8. PERFORMING ORGANIZATION REPORT NUMBER
VM-05-FINAL-REPORT

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DR. ROBERT HERKLOTZ
AFOSR/NM
4015 WILSON BLVD, ROOM 713
ARLINGTON, VA 22203-1954

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
N/A

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES
N/A

14. ABSTRACT
Report developed under STTR contract for topic AF04-T023. This Phase I project developed a modeling language and laid a foundation for computational support tools for specifying, analyzing, and verifying complex distributed system designs. Ultimately, the overall modeling and analysis framework will provide an integrated suite of tools and methods leading to qualitative improvements in the design of dependable distributed systems. In more detail, this project developed: (a) a formal modeling language, called TIOA (Timed Input/Output Automata), for specifying timed, asynchronous, and interacting systems components, (b) the front-end processor for TIOA, incorporating syntax and type checking, and providing interfaces to computer-aided design tools, (c) a simulation tool allowing simulation of specifications and paired simulations of a specification and an abstract implementation, and (d) a theorem-proving link through an interface to PVS. This project provides the basis for refining the language and extending and integrating the associated tool set in Phase II.

15. SUBJECT TERMS
STTR report, modeling language, distributed systems, simulation, specification, verification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			ALEXANDER A. SHVARTSMAN
U	U	U	SAR	58	19b. TELEPHONE NUMBER (Include area code) 860-486-2672

Final Report

A Framework for Modeling and Analyzing Complex Distributed Systems

Contract No. FA9550-04-C-0084

STTR Phase I

Nancy A. Lynch [1] Alexander A. Shvartsman [2]

August 15, 2005

Executive summary. This Small Business Technology Transfer Phase I project developed a modeling language and laid a foundation for computational support tools for specifying, analyzing, and verifying complex distributed system designs. Ultimately, the overall modeling and analysis framework will provide an integrated suite of tools and methods leading to qualitative improvements in the design of dependable distributed systems. In more detail, this project developed: (a) a formal modeling language, called TIOA (Timed Input/Output Automata), for specifying timed, asynchronous, and interacting systems components, (b) the front-end processor for TIOA, incorporating syntax and type checking, and providing interfaces to computer-aided design tools, (c) a simulation tool allowing simulation of specifications and paired simulations of a specification and an abstract implementation, and (d) a theorem-proving link through an interface to PVS. To demonstrate the feasibility of this approach, this project produced examples of specification and analysis of distributed algorithms using this framework. This project provides the basis for refining the language and extending and integrating the associated tool set in Phase II. This will include integration with model-checking tools, and longer term the development of tools enabling computer-aided generation of code from TIOA specifications.

For the reporting period from September 15, 2004 to July 15, 2005, the Phase I project team recorded the following accomplishments.

(1) The TIOA language definition document was produced.

(2) A Front End tool was developed to support the TIOA langauge and provide interfaces to other tools.

(3) A TIOA simulator was developed and a subset of TIOA was designed for use with the simulator.

(4) A prototype implementation of the translator from TIOA to PVS was developed and used with selected examples.

(5) Several examples were developed and used with the toolset. The examples use TIOA as the specification language. The example specifications were checked using the front end tool, and then selectively simulated using the TIOA simulator. The TIOA to PVS translator was used to produce native PVS specifications. Selected properties of the example specifications were verified using PVS.

(6) Ease-of-use features of the toolset have been considered, including a convenient graphical user interface. We also experimented with user-defined visualizations of simulated executions. Finally, an experimental partial port of the tools from Unix to Windows was explored.

(7) Internal project web repository was set up and used for all project documentation, specification, code, software, and examples.

The project closely adhered to the work documented in the original proposal.

[1]Principal Investigator and Chief Technology Officer, VEROMODO, Inc.. Email: lynch@theory.csail.mit.edu.

[2]Co-Principal Investigator and President, VEROMODO, Inc.. Email: aas@cse.uconn.edu.

Contents

1 Project Landscape and Summary of Accomplishments

During Phase I of our project we demonstrated the feasibility of providing a comprehensive computer-aided framework for modeling and analyzing complex distributed systems on the basis of a formally defined modelling language. In this section we describe the challenges we face and our approach for coping with them; we then overview our Phase I accomplishments, list personnel involved in the project, enumerate publications produced or finalized during the project, and overview the structure of the rest of the report.

1.1 Challenges in developing distributed systems.

Developing dependable distributed systems for modern computing platforms continues to be challenging. While the availability of distributed middleware makes feasible the construction of systems that run on distributed platforms, ensuring that the resulting systems satisfy specific safety, timing, and fault-tolerance requirements remains problematic. The middleware services used for constructing distributed software are specified informally and without precise guarantees of efficiency, timing, scalability, compositionality, and fault-tolerance. Even when services and algorithms are specified formally, rigorous reasoning about the specifications is often left out of the development process.

As contemporary distributed systems continue to grow in complexity and sophistication in many domains, these systems are required to have formally-specified guarantees of safety, performance, and fault-tolerance. Current software-engineering practice limits the specification of such requirements to informal descriptions. When formal specifications are given, they are typically provided only for the system interfaces. The specification of interfaces alone stops far short of satisfying the needs of users of critical systems. Such systems need to be equipped with precise specifications of their semantics and guaranteed behavior. When a system is built of smaller components, it is important to specify the properties of the system in terms of the properties of its components. We view formal specification and analysis as valuable tools that should be at the disposal of the developers of distributed systems.

1.2 Our approach to modeling and analysis of complex distributed systems.

Our approach uses mathematical models—in particular, interacting state machines—as an integral part of the software development process. The stages of this process within the scope of our framework are as follows. Abstract requirements for a distributed system are specified using a modeling language. These specifications are then refined through multiple levels of abstraction. Each refinement step is formally validated. Validation techniques include a combination of simulation, model checking, and theorem proving. The goal of the refinement process is to produce sufficiently detailed models that (a) can ultimately be used to generate distributed code automatically, and that (b) are guaranteed to be consistent with the modeled system requirements.

Our project builds on previous work by the investigators and others on the IOA language (named after Input/Output Automata) and accompanying toolset. This overall research direction was featured in the *2003 MIT Technology Review* as one of the *10 Emerging Technologies that will Change the World* [103]. IOA is a language for describing systems as compositions of interacting state machines; it supports proofs of invariants and simulation relationships between specifications at different levels of abstraction.

With the goal of providing a formal methodology and tools to substantially improve the state of the art in developing software for complex distributed systems, our current project includes the following: (i) development of the TIOA language (after Timed Input/Output Automata), which subsumes IOA and includes comprehensive facilities for modeling timing properties. (ii) development of commercial-grade automated tools supporting specification and analysis of distributed system designs in TIOA, including the language front end and a simulator, and (iii) development of interfaces to other computer-aided tools, such as model-checking and interactive theorem-proving tools. Longer-term, this development will lead to computer-aided generation of code from specifications.

To support automated formal methods for constructing or analyzing systems, a modeling language must rest on a solid mathematical foundation. The I/O automaton model [76] and its timed extensions [80, 74, 67, 42, 43, 40] provide such a foundation. I/O automata have been used to describe and verify many distributed algorithms and systems (see, for example, [65, 22, 17, 16, 21, 39, 97, 32, 73, 20]). Timed I/O Automata have been used to model timing-dependent distributed algorithms and real-time control systems [56, 107, 32, 84].

Timed I/O Automata are interacting state machines. They are nondeterministic, which makes them suitable for describing systems in their most general forms. The state of a TIOA can change in two ways: *discrete transitions*, which are labeled by discrete actions, change the state instantaneously, whereas *trajectories* are functions that describe the evolution of the state variables over intervals of time. An important feature of the TIOA framework is its support for decomposing system descriptions, either *horizontally* (into interacting components) or *vertically* (into multiple levels of abstraction). In particular, the framework includes a notion of *external behavior* for a Timed I/O Automaton, which describes the set of possibles interactions between the TIOA and its environment. The framework also defines what it means for one TIOA to *implement* another, based on an inclusion relationship between their external behavior sets, and defines a notion of *simulation*, which provides a sufficient condition for demonstrating implementation relationships. Timed I/O Automata can also be composed, using a *composition operation* that identifies external actions; this notion of composition respects external behavior. TIOAs admit a rich set of proof methods, including invariant assertion techniques for proving that a property is true in all reachable states, forward and backward simulation methods for proving that one automaton implements another, and compositional methods for reasoning about collections of interacting components.

Target systems. Many types of systems are currently developed using software engineering methodology that is less than adequate in its ability to handle formal modeling and analysis of complex distributed software, and we anticipate that several specific types of systems will benefit from being designed within our proposed framework. The types of systems include: (a) Distributed data systems: data collection, management, dissemination; consistent replicated shared-data systems. (b) Communication: group communication systems, broadcast and multicast systems with quality-of-service guarantees. (c) Coordination and control: traffic management, industrial process control, automated manufacturing systems, transportation (e.g., TCAS, traffic collision avoidance system used in civil aviation).

Many such systems involve specialized distributed platforms, such as networks of sensors and mobile *ad hoc* networks. We believe our approach will be an effective aid in solving a variety of common problems in these settings, for example, location determination, time synchronization, establishing communication structures (spanning trees, clusters, electing leaders, etc.), communication, data management, and tracking.

1.3 Phase I objectives and summary of accomplishments

We now summarize the original objectives for Phase I and our accomplishments. In the Phase I proposal [71], we identified the following three major objectives:

1. Design a modeling language that includes event ordering behavior and timing behavior. The language was to be based on the successful Input/Output Automata (IOA) language, which has proved to be very effective for modeling complex concurrent and distributed systems, for reasoning formally about system correctness, and for developing real implementations based on the IOA specifications. The target language, called Timed IOA, or TIOA, was to extend the IOA language to allow specification of timing behavior.
2. To enable the effective use of TIOA, we planned to prototype/integrate several tools that can be used by system designers to partially automate the design and verification of systems expressed in TIOA. We aimed to develop a *front end* processor for this language and to prototype tools to support *simulation* of specifications, including simulation at multiple levels of abstraction. We also planned to develop interfaces to *interactive proof* tools suitable for proving invariants and implementation relationships.
3. To demonstrate the feasibility and effectiveness of our approach, we planned to develop demonstrations

of the use of TIOA and the associated tools. This was to include complete examples of specification, refinement, and computer-aided verification of non-trivial systems.

We now summarize the accomplishments of the Phase I project.

- We have assembled a project team and set up a web-based tool (using Wiki) to coordinate the activities and to serve as the document repository. The repository contains project documentation, specification, code, software, and examples.
- We defined the TIOA language [42], which extends the highly successful IOA language [24]. The language is accompanied by a formal framework for reasoning about timed systems specified in this language [41, 40].
- We developed the front-end tool, which supports TIOA and interfaces to other computer-aided design and verification tools.
- We developed the TIOA Simulator, which is designed to work with a suitable subset of the TIOA language. The TIOA Simulator requires TIOA specifications to be augmented with scheduling specifications in order to resolve nondeterminism in the order in which events occur.
- We designed and documented an approach to translating TIOA specification to PVS specifications. We developed an initial implementation of a translator from TIOA to PVS and we used it with selected TIOA specification examples to reason about them using PVS.
- We developed a number of exploratory specification examples to use in conjunction with the front end, the simulator, the TIOA-to-PVS translator, and PVS. We are continuing to develop complete examples using TIOA and the toolset. These now include more complicated examples: a model for the SATS (Small Aircraft Transportation System) [19] and a model for the DHCPv.6 communication protocol [18].
- We have considered the ease-of-use features of the toolset. We also performed an experimental partial port of the tools from Unix to Windows.
- We prototyped a user-friendly graphical interface using Eclipse and we successfully explored the possibility of providing user-defined visualization of simulated executions.
- We have established collaborative relationship with some potential users of the work.

  Nancy Griffeth (CUNY/Lehman College) used TIOA to develop a virtually complete specification of DHCPv6.

  Myla Archer (Naval Research Lab) developed several proof strategies that were incorporated within our framework. She will be using TIOA in her work at NRL.
- We established contacts with researchers at Stony Brook University, Profs. Scott Smolka and Radu Grosu, with the goal of future cooperative development of the model checking component of our framework.

1.4 Personnel involved in Phase I work

The have assembled the following team that executed the Phase I project.

The structure of the project team as as follows:
Nancy Lynch, project technical leader and Co-PI
Alex Shvartsman, project manager and Co-PI
Peter Musial, application development
Steve Garland, front end and integration technical leader
Paul Attie, example development technical leader
Aleksandra Portnova, example development engineer

Dilsun Kaynar, simulator technical leader
Panayiotis Mavromatis, intermediate language development
Sajan Mitra, PVS integration
Shinya Umeno, TIOA/PVS example development
Vlad Lakin, Windows port feasibility study

Note that several team member were involved in multiple roles. Only the primary roles are given in the list above.

In order to enable the project team members to collaboratively (and remotely) edit a common set of web pages, specifications, and other documents, we set up a website that has a Wiki.

1.5 Publications

In this section we list publications directly related to the Phase I project that were authored or co-authored by the project personnel. All publications are available on request. One noteworthy item is the monograph [P9] that will be published in 2005 by Morgan-Clayton.

[P1] Fivos Constantinou, Dilsun Kaynar, and Panayiotis Mavrommatis, The TIOA Simulator. CSAIL Research Abstracts - 2005, http://publications.csail.mit.edu/abstracts/abstracts05/dilsun/dilsun.html

[P2] Stephen Garland, Dilsun Kaynar, Nancy Lynch, Joshua Tauber, and Mandana Vaziri, TIOA Tutorial, May 22, 2005

[P3] Nancy Griffeth and Nancy Lynch, TIOA Modeling for DHCP, October 27, 2004. http://tioa.csail.mit.edu/project/example-pages/dhcpv4/Article.pdf

[P4] Dilsun Kaynar, Nancy Lynch, and Sayan Mitra, Specifying and proving timing properties with TIOA tools. Work in Progress Session of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004), Lisbon, Portugal, Dec 2004.

[P5] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. The Theory of Timed I/O Automata. Revised and shortened version of Technical MIT-LCS-TR-917a (from 2004), March, 2005. http://theory.lcs.mit.edu/tds/papers/Kirli/TIOA-synthesis.ps

[P6] Joshua A. Tauber, Dilsun K. Kaynar, Nancy A. Lynch. Correctness of a Compiler for Distributed Algorithms. Submitted for publication, 2005.

[P7] Dilsun Kaynar, Nancy Lynch, Sayan Mitra, Christine Robson. Design for TIOA Modeling Language. Manuscript in progress, 2004. http://theory.lcs.mit.edu/ dilsun/TIOA.html

[P8] Dilsun Kaynar, Nancy Lynch, Sayan Mitra, and Stephen Garland, The TIOA Language, Design Notes, Version 0.21, May 22, 2005

[P9] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata.* Monograph to appear in Synthesis Series, Morgan-Claypool publishers, 2005 (also see Technical Report MIT-LCS-TR-917a, MIT Laboratory for Computer Science, Cambridge, MA, April, 2004, http://theory.lcs.mit.edu/tds/papers/Kirli/TIOA-tr-a.ps)

[P10] Dilsun K. Kaynar and Nancy A. Lynch. Decomposing Verification of Timed I/O Automata. *Formal Techniques, Modelling and Analysis of Timed and Fault Tolerant Systems: Joint International Conferences, FORMATS 2004, and Format Technicques in Real-Time and Fault-Tolerant Systems,* FTRTFT 2004, Grenoble, France, September 22-24, 2004. Volume 3253 of Lecture Notes in Computer Science, pages 84-101, Springer-Verlag, 2004.

[P11] Hongping Lim, Dilsun Kaynar, Nancy Lynch, and Sayan Mitra, Translating timed I/O automata specifications for theorem proving in PVS, To be published in the proceedings of *International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'05)*, Uppsala, Sweden, September 26-28, 2005

[P12] Hongping Lim, Nancy Lynch, and Sayan Mitra, Translating Timed I/O Automata Specifications for Theorem Proving in PVS. CSAIL Research Abstracts - 2005, http://publications.csail.mit.edu/abstracts/abstracts05/hongping/hongping.html

[P13] Nancy Lynch, Alex Shvartsman, A Framework for Modeling and Analyzing Complex Distributed Systems, August 10, 2004

[P14] N. Lynch and A. Shvartsman *Status Report 1: A Framework for Modeling and Analyzing Complex Distributed Systems, STTR Phase 1*, Contract FA9550-04-C-0084, VEROMODO, Inc., October 15, 2004.

[P15] N. Lynch and A. Shvartsman *Status Report 2: A Framework for Modeling and Analyzing Complex Distributed Systems, STTR Phase 1*, Contract FA9550-04-C-0084, VEROMODO, Inc., March 15, 2005.

[P16] Nancy Lynch and Shinya Umeno, Verification of the SATS Using IOA/TIOA: A Case Study. CSAIL Research Abstracts - 2005, http://publications.csail.mit.edu/abstracts/abstracts05/shinya/shinya.html

[P17] Sayan Mitra and Myla Archer. PVS Strategies for proving abstraction properties automata In *Electronic Notes in Theoretical Computer Science*, volume 125(2), 2005, pages 45-65.

(The complete bibliography cited in this report is included after the main text.)

1.6 Document structure.

In the rest of th report we focus on the details of the Phase I work and our accomplishments. We begin in Section 2 by providing a detailed introduction to the TIOA language, the cornerstone of our project. In Section 3 we describe the language front end tool and its interfaces. In Section 4 we present our work on the TIOA Simulator. In Section 5 we describe TIOA-to-PVS translation tool and the use of PVS in proving properties of TIOA specifications. We briefly discuss case studies in Section 6. We conclude the main text in Section 7. The appendices contain detailed information on selected cases studies and the overview of future planned work.

2 The TIOA Language

The timed input/output automata (TIOA) modeling framework [41, 40] is a mathematical framework that supports the description and analysis of timed systems. The TIOA language is a modeling language that provides notations for describing timed I/O automata precisely. The TIOA language is a variant of the IOA language [24], which can be used to describe basic I/O automata with no timing information. TIOA extends and formalizes the descriptive notations used in [41, 40] and supports a variety of analytic tools. These tools range from light-weight tools, which check the syntax of automaton descriptions, to medium-weight tools, which simulate the action of an automaton, and to heavier-weight tools, which provide support for proving properties of automata. Timed input/ouput automata provide a mathematical model suitable for describing time-dependent behavior in concurrent systems. The model provides a precise way of describing and reasoning about system components that interact with each other through discrete actions as well as the continuous evolution of internal state components over time.

2.1 Timed input/output automata

The fundamental object in the TIOA framework is a *timed (I/O) automaton*, which is a kind of nondeterministic, possibly infinite-state, state machine. The state of a timed automaton is described by a valuation of state variables that are internal to the automaton. The state of a timed automaton can change in two ways: instantaneously, by the occurrence of a *discrete transition*, or over an interval of time via a *trajectory*, which is a function that describes the evolution of the state variables. Trajectories may be continuous or discontinuous functions.

TIOA transitions are associated with named *actions*, which are classified as *input*, *output*, or *internal*. Input and output actions are used for communication with the automaton's environment, whereas internal actions are visible only to the automaton itself. The input actions are assumed not to be under the automaton's control, whereas the automaton itself controls which output and internal actions should be performed.

The communication of a timed automaton with its environment is limited to discrete transitions associated with actions shared between the automaton and its environment. The time domain in TIOA is the set of real numbers (in [40] additional time domains are considered). States of automata consist of valuations of *variables*. Each variable has both a *static type*, which defines the set of values it may assume, and a *dynamic type*, which gives the set of trajectories it may follow. We assume that dynamic types are closed under some simple operations: shifting the time domain, taking subintervals, and pasting together intervals.

A *trajectory* for a set V of variables describes the evolution of the variables in V over time; formally, it is a function from a time interval that starts with 0 to valuations of V; that is, a trajectory defines a value for each variable at each time in the interval.

Formally, a *timed (I/O) automaton* A consists of the following six components:

- A set X of *internal variables*.
- A set Q, which is a subset of all possible valuations of X.
- A set of initial states, which is a non-empty subset of the set of all states.
- A signature, which lists disjoint sets of input, output, and internal actions of A.
- A discrete transition relation, which contains triples of the form (state, action, state), and
- A set of trajectories for X such that $\tau(t) \in Q$ for every $\tau \in \mathcal{T}$ and every t in the domain of τ.

An action π is said to be *enabled* in a state s if there is another state s' such that (s, π, s') is a transition of the automaton. Input actions are enabled in every state; i.e., automata are not able to "block" input actions from occurring. The *external* actions of an automaton consist of its input and output actions.

The transition relation is usually described in *precondition-effect* style, which groups together all transitions that involve a particular type of action into a single piece of code. The precondition is a predicate (that is, a boolean-valued expression) on the state indicating the conditions under which the action is permitted to occur. The effect describes the changes that occur as a result of the action, either in the form of a simple program or in the form of a predicate relating the pre-state and the post-state (i.e., the states before and after the action occurs). Actions are executed indivisibly.

Trajectories are defined using invariants, algebraic and differential equations, and "urgency" conditions that specify when time must stop to allow a discrete action to occur.

2.2 Executions and traces

An *execution fragment* of an I/O automaton is either a finite sequence s_0, π_1, s_1, π_2, ..., π_n, s_n, or an infinite sequence s_0, π_1, s_1, π_2, ..., of alternating states s_i and actions π_i such that $(s_i, \pi_{i+1}, s_{i+1})$ is a transition of the automaton for every $i \geq 0$. An *execution* is an execution fragment that begins with a start state. A state is *reachable* if it occurs in some execution. The *trace* of an execution is the sequence of external actions in that execution.

2.3 Properties of automata

An *invariant* of an automaton is any property that is true in all reachable states of the automaton.

An automaton A is said to *implement* an automaton B provided that A and B have the same input and output actions and that every trace of A is a also trace of B. In order to show that A implements B, one can use a *simulation relation* between states of A and states of B such that, loosely speaking, every start state of A is related to a start state of B and every reachable state of A is related to a state of B reached by the same series of external actions.

For the purpose of a formal definition, we assume that A and B have the same input and output actions. A relation R between the states of A and B is a *forward simulation*[3] with respect to invariants I_A and I_B of A and B if and only if

- every start state of A is related (via R) to a start state of B, and
- for all states s of A and u of B satisfying the invariants I_A and I_B such that $R(s,u)$, and for every step (s,π,s') of A, there is an execution fragment α of B starting with u, containing the same external actions as π, and ending with a state u' such that $R(s',u')$.

A general theorem is that A implements B if there is a forward simulation from A to B.

Similarly, a relation R between the states of A and B is a *backward simulation*[4] with respect to invariants I_A and I_B of A and B if

- every state of A that satisfies I_A corresponds (via R) to some state of B that satisfies I_B,
- if a start state s of A is related (via R) to a state u of B that satisfies I_B, then u is a start state of B, and
- for all states s, s' of A and u' of B satisfying the invariants such that $R(s',u')$, and for every step (s,π,s') of A, there is an execution fragment α of B ending with u', containing the same external actions as π, and starting with a state u satisfying I_B such that $R(s,u)$.

Another general theorem is that A implements B if there is an *image-finite* backward simulation from A to B. Here, a relation R is image-finite provided that for any x there are only finitely many y such that $R(x,y)$. Moreover, the existence of any backward simulation from A to B implies that all finite traces of A are also traces of B.

2.4 Using TIOA to formalize descriptions of timed I/O automata

We illustrate the nature of timed automata, as well as the use of TIOA to define the automata, by a few simple examples. Figure 1 contains a simple TIOA description for an automaton, `Timeout(u:Real, M:Type)`, that awaits the receipt of a message of type `M` from another process. If `u` time units elapse without such a message arriving, the automaton performs a `timeout` action, thereby "suspecting" the other process. When a message arrives, it "unsuspects" the other process. The automaton may suspect and unsuspect repeatedly. The automaton is parameterized by the timeout period `u` and the type `M` of the messages received by `Timeout(u:Real, M: Type)`.

The automaton `Timeout` has two state variables: `suspected` is a boolean that is set to `true` when a `timeout` occurs, and `clock` is a real number that represents a timer running at the same speed as realtime. The initial value of `suspected` and `clock` are `false` and `0`. The value of the automaton parameter `u` is constrained to be strictly greater than `0`.

The transitions of the automaton `Timeout` are given in precondition/effect style. The input action `receive(m)` has no precondition, which is equivalent to having `true` as its precondition. This is the case for all input actions; that is, every input action in every automaton is enabled in every state. The effect of `receive` is to reset `clock` to `0` and to set `suspected` to false (in case it had been true before). The output action `timeout` can occur only when it is enabled, that is, only in states in which `suspected` is false and `clock = u`. Its effect is to set `suspected` to `true`.

[3] In some previous work such relations are called *weak* forward simulations.

[4] In some previous work such relations are called *weak* backward simulations.

```
automaton Timeout(u: Real, M: type) where u > 0
  signature
    input receive(m: M)
    output timeout
  states
    suspected: Bool := false,
    clock Real := 0
  transitions
    input receive(m)
      eff clock := 0;
          suspected:= false
    output timeout
      pre ¬suspected ∧ clock = u
      eff suspected := true
  trajectories
    trajdef suspected
      invariant suspected
      evolve d(clock) = 1
    trajdef notsuspected
      invariant ¬suspected
      stop when clock = u
      evolve d(clock) = 1
```

Figure 1: TIOA description of a timeout process

The two trajectory definitions `suspected` and `notsuspected` correspond to two "modes" of the `Timeout` automaton. While the suspected flag is `false`, the clock advances with rate `1`, that is, with the same rate as realtime, and time cannot go beyond the point at which `clock = u`. While the suspected flag is `true` there is no condition on time-passage; the clock may keep advancing with rate `1`. Note that trajectories do not need to be followed until a stopping condition is reached; however, if a stopping condition is reached then time must stop. At this point, a discrete action may occur if it is enabled.

2.5 Data types in TIOA descriptions

TIOA enables users to define the actions and states of I/O automata abstractly, using mathematical notations for sets, sequences, etc., without having to provide concrete representations for these abstractions. Some mathematical notations are built into TIOA; others can be defined by the user. The data types `Bool`, `Int`, `Nat`, `Real`, `Char`, and `String` can appear in TIOA descriptions without explicit definition. Compound data types can be constructed using the following type constructors and used without explicit definition.

- `Array[I1, ..., In, E]` is an n-dimensional array of elements of type `E` indexed by elements of types `I1`, ..., `In`.
- `Map[D1, ..., Dn, R]` is a finite partial mapping of elements of an n-dimensional domain with type `D1` $\times \cdots \times$ `Dn` to elements of a range with type `R`. Mappings differ from arrays in that they are defined only for finitely many elements of their domains (and hence may not be totally defined).
- `Seq[E]` is a finite sequence of elements of type `E`.
- `Set[E]` is a finite set of elements of type `E`.
- `Mset[E]` is a finite multiset of elements of type `E`.
- `Null[E]` is isomorphic to `E` extended by a single element `nil`.

In this tutorial, we describe operators on the built-in data types informally when they first appear in an example.

Users can introduce additional data types and type constructors by defining *vocabularies* for them. Each vocabulary introduces notations for a set of types and a set of operators. In fact, each of the built-in data types is defined by a built-in vocabulary. For example, the following built-in vocabularies provides notations for the `Real` data type and its associated operators. Each operator has a *signature* that specifies the types of its arguments and the type of its result. Infix, prefix, postfix, and mixfix operators are named by sequences of non-letter characters and are defined using placeholders __ to indicate the locations of their arguments. Operators used in functional notation (e.g., in $max(a, b)$) are named by simple identifiers.

Logical Operator			Datatype Operator		
Symbol	Meaning	Input	Symbol	Meaning	Input
$\forall$	For all	\A	$\leq$	Less than or equal	<=
$\exists$	There exists	\E	$\geq$	Greater than or equal	>=
$\neg$	Not	~	$\in$	Member of	\in
$\neq$	Not equals	~=	$\notin$	Not a member of	\notin
$\wedge$	And	/\	$\subset$	Proper subset of	\subset
$\vee$	Or	\/	$\subseteq$	Subset of	\subseteq
$\Rightarrow$	Implies	=>	$\supset$	Proper superset of	\supset
$\Leftrightarrow$	If and only if	<=>	$\supseteq$	Superset of	\supseteq
			$\vdash$	Append element	|-
			$\dashv$	Prepend element	-|

Table 1: Typographical conventions

```
vocabulary Real
  imports NumericOps(type Real, type Real, type Real)
  operators
    -__, abs: Real → Real
    __**__: Real, Int → Real
    int2real: Int → Real
vocabulary NumericOps(T1, T2, T3: type)
  types T1 T2 T3
  operators
    __+__ , __-__ , __*__ , __/__ , min, max: T1, T2 → T3
    __<__ , __≤__ , __>__ , __≥__ , __=__ , __≠__: T1, T2 → Bool
```

As these examples illustrate, a vocabulary can import notations from another vocabularies, and it can be parameterized to make operator notations such as `__<__:Real,Real→Bool` available for the `Real` data type.

A vocabulary can define a type constructor, as in the following built-in vocabulary for the `Null` constructor.

```
vocabulary Null defines Null[T]
  operators
    nil : → Null[T]
    embed : T → Null[T]
    __.val : Null[T] → T
```

The identifier `T` in this vocabulary is a type parameter, which is instantiated any time the constructor `Null` is used to provide operator notations appropriate for that use. Thus, if `x` is a variable of type `Null[Int]`, then one can write `embed(x).val = x`.

User-defined vocabularies can introduce notations for enumeration, tuple, and union types analogous to those found in many common programming languages. For example,

```
vocabulary sampleVocab
  types Color enumeration [red, white, blue],
        Msg   tuple [source, dest: Process, contents: String],
        Fig   union [sq: Square, circ: Circle]
```

can be imported by the definition of any other vocabulary or automaton to provide notations for three data types it describes.

In this tutorial, some operators are displayed using mathematical symbols that do not appear on the standard keyboard. Table 1 shows the input conventions for entering these symbols.

2.6 TIOA descriptions for primitive automata

Explicit descriptions of primitive automata specify their names, action signatures, state variables, transition relations, and trajectories. All but the last of these elements must be present in every primitive automaton description.

2.6.1 Automaton names and parameters

The first line of an automaton description consists of the keyword **automaton** followed by the name of the automaton. As illustrated in Figure 1, the name may be followed by a list of formal parameters enclosed within parentheses. There are two kinds of automaton parameters. An individual parameter (such as `u:Real`) consists of an identifier with its associated type, and it denotes a fixed element of that type. A type parameter (such as `M:`**type**) consists of an identifier followed by the keyword **type**, and it denotes a type.Example of nondeterministic choice of initial value for state variable

2.6.2 Action signatures

The signature for an automaton is declared using the keyword **signature** followed by lists of entries describing the automaton's input, internal, and output actions. Each entry contains a name and an optional list of parameters enclosed in parentheses. There are two kinds of action parameters. A varying parameter consists of an identifier with its associated type, and it denotes an arbitrary element of that type. A fixed parameter consists of the keyword **const** followed by a term denoting a fixed element of its type. Neither kind of parameter can have **type** as its type. Each entry in the signature denotes a set of actions, one for each assignment of values to its varying parameters.

It is possible to constrain the values of the varying parameters for an entry in the signature using the keyword **where** followed by a predicate. Such constraints restrict the set of actions denoted by the entry.

2.6.3 State variables

As in the above examples, state variables are declared using the keyword **states** followed by a comma-separated list of state variables and their types. State variables can be initialized using the assignment operator $:=$ followed either by an expression or by a nondeterministic choice. The order in which state variables are declared makes no difference: state variables are initialized simultaneously, and their initial values cannot refer to the value of any state variable.

A nondeterministic choice (of the form **choose** *variable* **where** *predicate*) selects an arbitrary value of the variable that satisfies the predicate. When a nondeterministic choice is used to initialize a state variable, there must be some value of the variable that satisfies the predicate. If the predicate is true for all values of the variable, then the effect is the same as if no initial value had been specified for the state variable.

```
automaton Choice
  signature output result(i: Int)
  states num: Int := choose n where 1 ≤ n ∧ n ≤ 3
  transitions
    output result(i)
      pre i = num
```

Figure 2: Example of nondeterministic choice of initial value for state variable

For example, in the automaton `Choice` (Figure 2), the state variable `num` is initialized nondeterministically to some value `n` that satisfies the predicate $1 \leq n \wedge n \leq 3$, i.e., to one of the values `1`, `2`, or `3` (the value of `n` must be an integer because it is constrained to have the same type, `Int`, as the variable `num` to which it will be assigned).

It is also possible to constrain the initial values of all state variables taken together, whether or not initial values are assigned to any individual state variable. This can be done using the keyword **initially** followed by a predicate (involving state variables and automaton parameters), as illustrated by the definition of the automaton `Shuffle` in Figure 3.[5]

```
vocabulary cardDeck
  types cardIndex enumeration [1, 2, 3, ..., 52]

automaton Shuffle
  imports cardDeck
  signature
    internal swap(i, j: cardIndex)
    output   deal(a: Array[cardIndex, String])
  states
    dealt: Bool := false,
    name:  Array[cardIndex, String],
    cut:   cardIndex
    initially
      ∀ i: cardIndex (i ≠ 52 ∧ i ≠ cut ⇒ name[i] < name[succ(i)])
            ∧ name[52] < name[1]
  transitions
    internal swap(i, j; local temp: String)
      pre ¬dealt
      eff temp := name[i];
          name[i] := name[j];
          name[j] := temp
    output deal(a)
      pre ¬dealt ∧ a = name
      eff dealt := true
```

Figure 3: Example of a constraint on initial values for state variables

In Figure 3, the initial values of the variable `cut` and the array `name` of strings are constrained so that `name[1], ..., name[52]` are sorted in two pieces, each in increasing order, with the piece after the `cut` containing smaller elements than the piece before the `cut`. The constraint following **initially** limits only the initial values of the state variables, not their subsequent values. (Note that the scope of the **initially** clause is the entire set of state variable declarations.) The type `Array[cardIndex, String]` of the state variable `name` consists of arrays indexed by elements of type `cardIndex` and containing elements of type `String` (see Section 2.5). The `swap` actions transpose pairs of strings until a `deal` action announces the contents of the array; then no further actions occur.

When the type of a state variable is an `Array`, `Map`, or **tuple** (Section 2.5), TIOA also treats the elements of the array or mapping, or the fields in the tuple, as state variables, to which values can be assigned without affecting the values of the other elements in the array or mapping or of the fields in the tuple.

2.6.4 Transition relations

Transitions for the actions in an automaton's signature are defined following the keyword **transitions**. A transition definition consists of an action type (i.e., **input**, **internal**, or **output**), an action name with optional parameters, an optional **where** clause, an optional precondition, and an optional effect.

[5] At present, users must expand the ... in the definition of the type `cardIndex` by hand. In the future, TIOA may provide more convenient notations for integer subranges.

More than one transition definition can be given for an entry in an automaton's signature. For example, the transition definition for the `swap` actions in the `Shuffle` automaton (Figure 3) can be split into two parts:

```
internal swap(i, j; local temp: String) where i ≠ j
  pre ¬dealt
  eff temp := name[i];
      name[i] := name[j];
      name[j] := temp
internal swap(i, i)
  pre ¬dealt
```

The second of these two transition definitions does not change the state, because it has no **eff** clause.

2.6.4.1 Transition parameters

Two kinds of parameters can be specified for a transition: ordinary parameters, corresponding to those in the automaton's signature, and additional local parameters. The ordinary parameters accompanying an action name in a transition definition must match those accompanying the action name in the automaton's signature, both in number and in type. However, the keyword **const** does not appear in transition parameters, and all transition parameters are treated as terms.

The simplest way to formulate the ordinary parameters for an action in a transition definition is to erase the keyword **const** and the type modifiers from the parameter list in the signature;

In addition to these ordinary parameters, a transition definition can contain *local variables*, which are specified after the ordinary parameters and the keyword **local**. Local variables can be used for two purposes. As illustrated in Figure 3, they can be used as temporary state variables. In addition, they can be used to relate the postcondition for a transition to its precondition.

2.6.4.2 Preconditions

A precondition can be defined for a transition of an output or internal action using the keyword **pre** followed by a predicate. Preconditions cannot be defined for transitions of input actions. All variables in a precondition must be parameters of the automaton, be state or local variables, appear in parameters for the transition definition, or be quantified explicitly in the precondition. If no precondition is given, it is assumed to be `true`.

An action is said to be *enabled* in a state if there are some values for the local variables of one of its transition definitions that satisfy both the **where** clause and, together with the state variables, the precondition for that transition definition. Input actions, whose transitions have no preconditions, are always enabled.

2.6.4.3 Effects

The effect of a transition, if any, is defined following the keyword **eff**. This effect is generally defined in terms of a (possibly nondeterministic) program that assigns new values to state variables. The amount of nondeterminism in a transition can be limited by a predicate relating the values of state variables in the post-state to each other and to their values in the pre-state.

If the effect is missing, then the transition has none; i.e., it leaves the state unchanged.

Using programs to specify effects A program is a list of statements, separated by semicolons. Statements in a program are executed sequentially. There are three kinds of statements: (1) *assignment statements*, (2) *conditional statements*, and (3) **for** *statements*.

Assignment statements An assignment statement changes the value of a state or local variable. The statement consists of the state or local variable followed by the assignment operator $:=$ and either an expression or a nondeterministic choice (indicated by the keyword **choose**). As noted in Section 2.6.3, the elements in an array or

mapping, or the fields in a tuple, used as a state or local variable, are themselves considered as separate variables and can appear on the left side of the assignment operator.

The expression or nondeterministic choice in an assignment statement must have the same type as the state or local variable. The value of the expression is defined mathematically, rather than computationally, in the state before the assignment statement is executed.[6] The value of the expression then becomes the value of the state or local variable in the state after the assignment statement is executed. Execution of an assignment statement does not have side-effects; i.e., it does not change the value of any state or local variable other than the one on the left side of the assignment operator.

Conditional statements A conditional statement is used to select which of several program segments to execute in a larger program. It starts with the keyword **if** followed by a predicate, the keyword **then**, and a program segment; it ends with the keyword **fi**. In between, there can be any number of **elseif** clauses (each of which contains a predicate, the keyword **then**, and a program segment), and there can be a final **else** clause (which also contains a program segment).

For *statements* A **for** statement is used to perform a program segment once for each value of a variable that satisfies a given condition. It starts with the keyword **for** followed by a variable, a clause describing a set of values for this variable, the keyword **do**, a program segment, and the keyword **od**.

```
vocabulary Packet
  types Message, Node,
        Packet tuple [contents: Message, source: Node, dest: Set[Node]]

automaton Multicast
  imports Packet
  signature
    input    mcast(m: Message, i: Node, I: Set[Node])
    internal deliver(p: Packet)
    output   read(m: Message, j: Node)
  states
    network: Mset[Packet] := {},
    queue:   Array[Node, Seq[Packet]]
    initially ∀ i: Node (queue[i] = {})
  transitions
    input mcast(m, i, I)
      eff network := insert([m, i, I], network)
    internal deliver(p)
      pre p ∈ network
      eff for j: Node in p.dest do queue[j] := queue[j] ⊢ p od;
          network := delete(p, network)
    output read(m, j)
      pre queue[j] ≠ {} ∧ head(queue[j]).contents = m
      eff queue[j] := tail(queue[j])
```

Figure 4: Example showing use of a **for** statement

Figure 4 illustrates the use of a **for** statement in a high-level description of a multicast algorithm that has no timing constraints. Its first line defines the `Packet` data type to consist of triples `[contents, source, dest]`, in which `contents` represents a message, `source` the `Node` from which the message originated, and `dest` the

[6] If a program consists of more than a single assignment statement, then the states before and after the assignment statements in the program may be intermediate states, which do not appear in the execution fragments of the automaton.

set of Nodes to which the message should be delivered. The state of the multicast algorithm consists of a multiset network, which represents the packets currently in transit, and an array queue, which represents, for each Node, the sequence of packets delivered to that Node, but not yet read by the Node.

The mcast action inserts a new packet in the network; the notation [m, i, I] is defined by the tuple data type (Section 2.5) and the insert operator by the multiset data type (Section 2.5). The deliver action, which is described using a **for** statement, distributes a packet to all nodes in its destination set (by appending the packet to the queue for each node in the destination set and then deleting the packet from the network). The read action receives the contents of a packet at a particular Node by removing that packet from the queue of delivered packets at that Node.

There are two ways to describe the set of values for the control variable in a **for** statement. The first consists of the keyword **in** followed by an expression denoting a set or multiset of values of the appropriate type, in which case the program following the keyword **do** is executed once for each value in the set or multiset. The second consists of the keyword **where** followed by a predicate, in which case the program is executed once for each value satisfying the predicate. These executions of the program occur in an arbitrary order. However, TIOA requires that the effect of a **for** statement be independent of the order in which executions of its program occur.

Using predicates on states to specify effects The results of a program can be constrained by a predicate relating the values of state variables after a transition has occurred to the values of state variables before the transition began. Such a predicate is particularly useful when the program contains the nondeterministic **choose** operator. For example,

```
eff name[i] := choose;
    name[j] := choose
    ensuring name'[i] = name[j] ∧ name'[j] = name[i]
```

is an alternative way of writing the effect clause of the swap action in Shuffle (Figure 3). The assignment statements indicate that the array name may be modified at indices i and j, and the **ensuring** clause constrains the modifications. A primed state variable in this clause (i.e., name') indicates the value of the variable in the post-state; an unprimed state variable (i.e., name) indicates its value in the pre-state. This notation allows us to eliminate the local variable temp needed previously for swapping.

There are important differences between **where** clauses attached to nondeterministic **choose** operators and those attached to **ensuring** clauses. A **where** clause restricts the value chosen by a **choose** operator in a single assignment statement, and variables appearing in the **where** clause denote values in the state before the assignment statement is executed. An **ensuring** clause can be attached to an entire **eff** clause; unprimed variables appearing in an **ensuring** clause denote values in the state before the transition represented by the entire **eff** clause occurs, and primed variables denote values in the state after the transition has occurred.

2.6.5 Trajectories

Trajectories of an automaton are defined following the keyword **trajectories**. A trajectroy definition consists of a name, an invariant, an **evolve** clause, and a stopping condition. More than one trajectory definition can be used to define trajectories of an automaton. For example, the automaton Timeout in Figure 1 has two trajectory definitions.

Each trajectory definition defines a set of trajectories; the set of all trajectories for an automaton is the concatenation closure of all of these sets (see [40] for the definition of concetanation for trajectories).A trajectory belongs to the set of trajectories defined by a trajectory definition if it satisfies the predicate in its **invariant** clause, the differential equations in the **evolve** clause and the stopping condition expressed by the **stop when** clause. The stopping condition is satisfied by a trajectory if the only state in which the condition holds is the last state of that trajectory. In other words, time cannot advance once the stopping condition becomes true.

The algorithm ClockSync is based on the exchange of physical clock values between different processes in the system. The parameter u determines the frequency of sending messages. Processes in the system are

```
automaton ClockSync(u,r: Real, i: Index)
  signature
    input receive(m: Real, j: Index, const i: Index) where j ≠ i,
    output send(m: Real, const i: Index)
  states
    nextsend: Real := 0,
    maxother: Real := 0,
    physclock: Real := 0
    initially u > 0 ∧ (0 ≤ r < 1)

  let logclock = max(maxother, physclock)

  transitions
    input receive(m,j,i)
      eff maxother := max(maxother,m)
    output send(m,i)
      pre m = physclock ∧ physclock = nextsend
      eff nextsend := nextsend + u
  trajectories
    trajdef always
      stop when physclock = nextsend
      evolve (1 - r) ≤ d(physclock) ≤ (1 + r)
```

Figure 5: Example showing trajectory definitions

indexed by the elements of the type `Index` which we assume to be pre-defined. `ClockSync` has a physical clock `physclock`, which may drift from the real time with a drift rate bounded by `r`. It uses the variable `maxother` to keep track of the largest physical clock value of the other processes in the system. The variable `nextsend` records when it is supposed to send its physical clock to the other processes. The logical clock, `logclock`, is defined to be the maximum of `maxother` and `physclock`. Formally `logclock` is a *derived variable*, which is a function whose value is defined in terms of the state variables.

The unique trajectory definition in this example shows that the variable `physclock` drifts with a rate that is bounded by `r`. The periodic sending of physical clocks to other processes is enforced through the stopping condition in the trajectory specification. Time is not allowed to pass beyond the point where `physclock = nextsend`.

3 Language Front-End and Interfaces

The front end enables the mathematical TIOA language to be used with computer-aided design tools. This includes the ability to parse a program specified in TIOA and to provide a semantics of the program relevant to the computer-aided tools.

In Phase I we developed a preliminary version of a front end for TIOA supporting specification, static, syntactic, and type analysis of specifications, and providing interfaces to computer-aided design tools. The front end is available on the (internal) website. The tool handles much of the TIOA language, including transition definitions with stopping conditions as well as trajectory definitions. (Here, "handles" means parses, performs static checks, and prettyprints.) Additional features include definitions for derived variables, notations for new NDR (nondeterminism resolution) constructs, and intermediate-language output.

The Intermediate Language (IL) facilitates interfacing between specifications expressed in TIOA and the input formats required by computer-aided tools, such as PVS, the simulator, and (planned for Phase II) a model checker.

To summarize, there are three targets that will be serviced by the TIOA front-end:

1. Simulation (prototyped in Phase I),

2. Model checking (front end "hooks" in Phase I, full integration in Phase II),
3. Theorem proving (prototyped in Phase I).

3.1 `tioaCheck`: a static semantic checker for TIOA

`tioaCheck` is a front-end checker for the syntax and static semantics of TIOA specifications. It can be invoked from the command line by typing

```
tioaCheck [option ...] source-file ...
```

or it can be invoked under Eclipse to check the TIOA specification displayed in an editor window. The names of the source files containing TIOA specifications must end with `.tioa`.

If no options are provided on the command line, `tioaCheck` simply reports any errors that it finds in the source files. For example, given a source file containing the specification

```
automaton A
  signature input get(n: Int) output put(n: Int)
  states value: Int
  transitions
    input get(n)
      eff value := f(n)
    output put(n)
      pre n = value
      eff n := n + 1
```

`tioaCheck` reports two errors:

```
6:20 undeclared 1-ary operator 'f'
9:11 cannot assign to 'n'
```

If options are provided, they produce the following results.

- `-p` causes `ioaCheck` to prettyprint (on the standard output) the selected TIOA source files. Prettyprinting indents TIOA specifications to reveal their structure, and it breaks lines that exceeds the margin. Prettyprinting via `tioaCheck` does not highlight keywords. Instead, users wishing to display TIOA specifications in a LaTeX document can use a special TIOA style file, which preserves line breaks and spacing, but highlights keywords and replaces notations such as `\in` with their mathematical counterparts (e.g., $\in$).

- `-il` causes `ioaCheck` to translate the selected TIOA source files into the TIOA Intermediate Language and to write the result on the standard output.

3.2 Front end functionality development

`tioaCheck` extends, simplifies, and differs from our previously developed `ioaCheck` tool (for checking specifications for untimed automata) in the following respects.

- It recognizes TIOA notations for time-related aspects of timed I/O automata: both discrete and continuous types, stopping conditions for transitions, trajectories, timing-enhanced schedules for the simulator, and timing-enhanced simulation relation definitions.
- It defines notations for operators on datatypes using the new `vocabulary` construct. In IOA, such notations were defined in auxiliary files, written in the Larch Shared Language (LSL). By eliminating the dependence of TIOA on LSL, we obtain two advantages:

  - Complete TIOA specifications can be written in a single file and can be checked statically without the need for auxiliary files written in LSL. (TIOA specifications can still be distributed over several files, if the user wishes).
  - `tioaCheck` and TIOA are more amenable than `ioaCheck` and IOA for use with theorem provers (e.g., PVS) other than the Larch Prover. They allow axioms for abstract data types to be expressed in languages appropriate for those theorem provers, rather than requiring them to be written in LSL.
  - `tioaCheck` recognizes the new, uniform parameterization mechanism in TIOA for definitions of automata and vocabularies.

- `tioaCheck` recognizes and checks the new notations in TIOA for defining functions and derived variables. It also recognizes and checks notations for new NDR (nondeterminism resolution) constructs that provide scheduling information for the TIOA simulator (see Section 4).
- `tioaCheck` is written entirely in Java 5.0. This makes it more transportable than `ioaCheck`, which was more difficult to compile because it was written in a combination of Java 1.4 and PolyJ (an extension to Java 1.4 that provided the benefits of generic types before they were introduced into Java 5.0).

3.3 Front end graphical user interface

Our ultimate goal for the user interface is to take advantage of features provided by the Eclipse Rich Client Platform development platform (see http://www.eclipse.org/rcp). In Phase I, lower level code improvements take take advantage of features provided by the Eclipse Java Development Tools (see http://www.eclipse.org/jdt).

We have prototyped a plug-in for the Eclipse platform that enables TIOA Syntax highlighting (Figure 3.3). This facilitates editing TIOA specifications by using Eclipse features for content assistance (e.g., keyword and identifier completion), formatting, and refactoring. We established the feasibility of improvements in error and warning messages. This is done by convert the messages to Eclipse format with problem markers. This will enable us in the future (Phase II) to provide suggestions to users and mechanisms for resolving errors. Additionally we can facilitate reading and editing TIOA specifications by providing Eclipse-based browsing features for locating declarations and references.

4 The TIOA Simulator

The TIOA Simulator is a powerful tool for running TIOA programs on a single machine and observing their behavior. It can be used to test the programs for bugs or problems, lead to proof strategies and provide performance predictions. The Simulator provides methods for resolving the nondeterminism of TIOA and for specifying paired simulations. After an overview of the purpose of simulation, we present the design and implementation of the TIOA Simulator prototype.

4.1 Purpose of simulation

The ability to simulate TIOA programs is particularly useful for a number of reasons. First, the simulator can test the proposed model of the system and reveal any possible problems with it. Furthermore, simulations can provide a better understanding of the system, and help develop strategies about proofs of correctness. Finally, the simulator's trace can be used to extract possible performance predictions. We further analyze the purpose of simulation below.

Simulation can test the model for a complex distributed system, and reveal any possible problems with the model. The Simulator "runs" the model on a single machine, tests certain assertions and outputs the trace of the execution. Testing the model with simulation can reveal implementation bugs or specification problems. If, on the other hand, all the simulated executions are correct, more confidence is gained that the model is also correct.

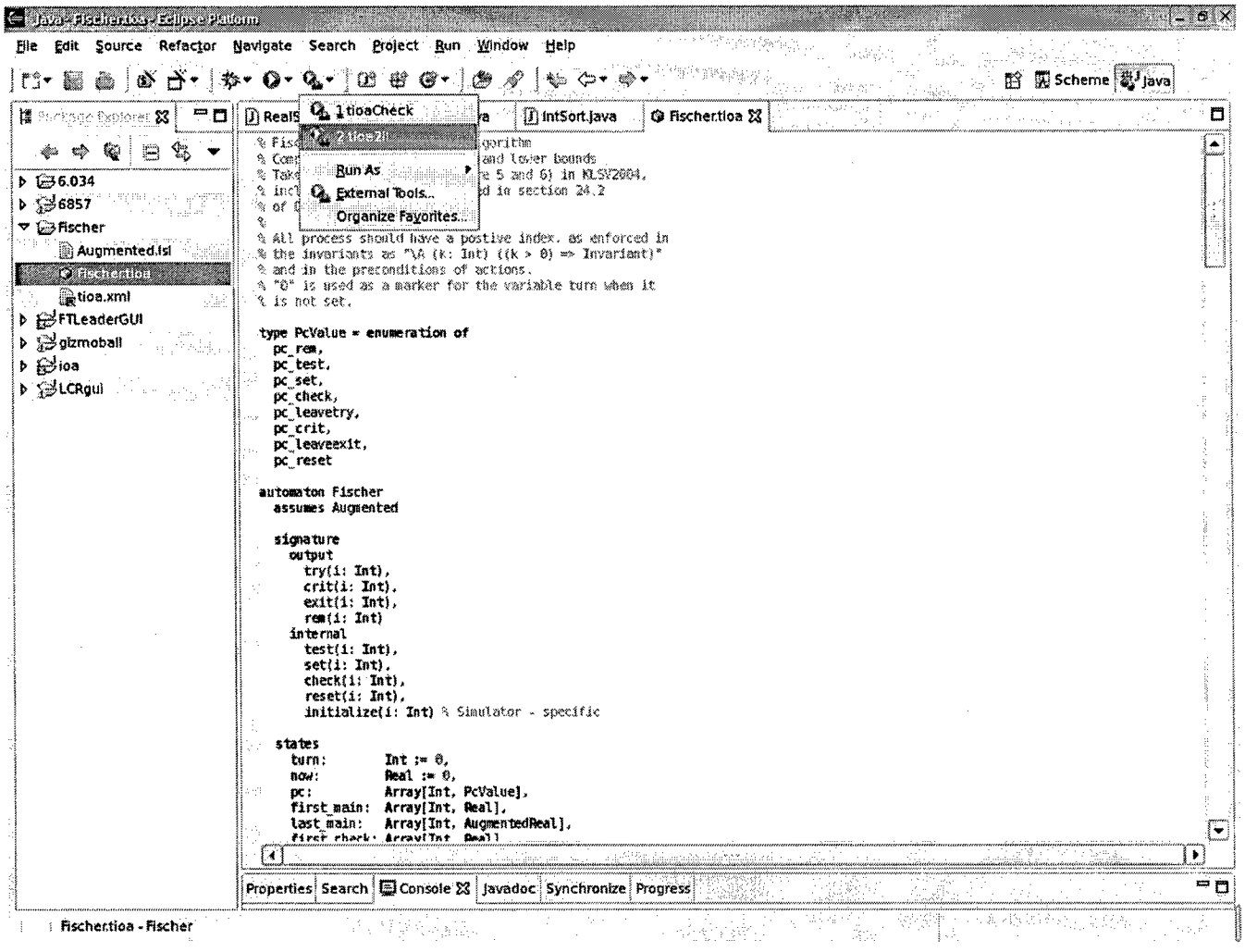

Figure 6: TIOA Language in the Eclipse Environment

Simulation not only leads to a better understanding of the system, but it can also be very helpful in developing strategies for proofs of correctness. First, the simulator allows discovering invariants and testing whether invariants hold throughout the sample executions. These invariants constitute good candidates for useful lemmas in later proofs of correctness. Second, the *paired simulator* allows simultaneous execution of two TIOA programs, the detailed, implementation program and a more abstract, specification automaton. Running the two together and automatically testing whether some invariants hold can lead to candidate *simulation relations*, which are frequently used proof techniques in distributed systems.

Even though the simulator runs on a single machine and does not target quantitative performance analysis, several measurements can be taken, such as the number of messages exchanged, or the number of steps taken before the completion of a specific task. The simulator therefore can be used to generate predictions about the system's performance.

4.2 Design Overview

Ideally, any TIOA program could automatically run through the Simulator without any modifications. There are, however, three main sources of problems that can be hard to solve automatically in general: (1) differential equations in **evolve** predicates and existential and universal quantifiers in other predicates (2) nondeterminism, either

from the scheduling of actions and trajectories or from **choose** statements, and (3) data type implementations from the abstract data type definitions.

Our solution to the above problems is to either restrict the language or let the programmer solve these problems on a case-by-case basis. In particular, we (1) slightly restrict TIOA to programs that can be simulated, (2) extend TIOA with syntax that can be used to resolve nondeterminism, and (3) provide a number of data type implementations and a way for the user to create implementations for new data types. Finally, we provide a way for the user to specify simulation relations and execute pairwise simulations. These topics are further analyzed below.

Restricting TIOA The TIOA simulator admits a subset of the TIOA language as its specification language, the TIOA-sim language. TIOA-sim imposes some restrictions on the trajectory definitions and on quantifiers. In particular, we restrict the form of differential equations in **evolve** clauses and the predicates that are used in stopping conditions so that the simulator can compute the value of a `Real` variable that is reached as a result of following a trajectory and detect the violation of stopping conditions. Moreover, no existential or universal quantifiers are allowed in TIOA-sim, unless the quantified variables are of an enumeration type. This restriction is necessary to allow automatic evaluation of predicates. Finally, the only **for** loops allowed in TIOA-sim are those over finite sets.

Resolving Nondeterminism Specifications in TIOA can incorporate several forms of nondeterminism. Automatically resolving nondeterminism is in general a hard problem. Instead, the TIOA Simulator provides a mechanism for resolving nondeterminism by letting the programmer specify which choice is made at any point.

Implementing Data Types TIOA provides syntax for specifying new (non-built-in) data types and operators. Other properties of the data types can also be specified in TIOA. However, implementing these data types only from the operators and the properties is impossible in the general case. The programmer is therefore asked to implement data types and their operators in Simulator's implementation language (Java), and instruct the Simulator to find these implementations. The Toolkit provides a large number of standard data types, from Integer, Real, String, to more complex data types such as Map, Array and Binary Search Tree, which are sufficient for most situations, and can be easily modified to fit more specific purposes.

Paired Simulations The TIOA simulator can be used to derive and test *simulation relations*. A simulation relation from an automaton A to another automaton B is specified as an assertion relating the states of A and B, when the two automata start on the same inputs and run with the same external behavior. For this purpose, the TIOA Simulator allows the user to specify this assertion and provide a set of step correspondences. The Simulator "runs" the two automata together, checks that the assertion holds and that the external behavior of the two automata is the same.

4.3 Restricting TIOA

We provide a formal description of the TIOA Simulator specification language (TIOA-sim) in Appendix C.1. Insight into the language can be gained through a simple example.

Example 4.1 The automaton `TwoTaskRace` below increments a counter until it is interrupted by a `set` action. After being interrupted it starts decrementing the counter and reports when the counter reaches 0. The timing constraints are expressed in absolute terms: (1) The variables `firstmain` and `lastmain` record the first and the last time that an action from the set {`increment`, `decrement`, `report`} is allowed to occur. (2) The variables `firstset` and `lastset` record the first and last time that the action `set` is allowed to occur. Note here that the actions of `TwoTaskRace` should be viewed as belonging to two separate tasks, each of which is subject to separate timing constraints. The first task consists of the actions `increment`,`decrement`, and `report` with

lower and upper bounds `a1` and `a2`, respectively, while the second one consists of the action `set`, with lower and upper bounds `b1` and `b2`, respectively.

The specification of `TwoTaskRace` below is a well-formed TIOA-sim program. The variable `now` is the single non-discrete variable of type `Real`. There are two trajectory definitions `preset` and `postset`, each of which has an **evolve** clause with `d(now)=1` and the stopping conditions are of the required form. In stopping conditions `now` is compared to variables of type `discrete Real`.

```
automaton TwoTaskRace(a1,a2,b1,b2 : Real)
  signature
    internal increment, decrement, set
    output report
  states
    count:      Nat := 0,
    flag:       Bool := false,
    reported:   Bool := false,
    now:        Real := 0,
    firstmain:  DiscreteReal := a1,
    lastmain:   discrete AugmentedReal := a2,
    firstset:   discrete AugmentedReal := b1,
    lastset:    discrete AugmentedReal := b2
  transitions
    internal increment
      pre ¬flag ∧ now ≥ firstmain
      eff count := count + 1;
          firstmain := now + a1;
          lastmain := now + a2
    internal set
      pre ¬flag ∧ now ≥ firstset
      eff flag := true;
          firstset := 0;
          lastset := infty
    internal decrement
      pre flag ∧ count > 0 ∧ now ≥ firstmain
      eff count := count - 1;
          firstmain := now + a1;
          lastmain := now + a2
    output report
      pre flag ∧ count = 0 ∧ ¬reported
                ∧ now ≥ firstmain
      eff reported := true;
          firstmain := 0;
          lastmain := infty
  trajectories
    trajdef preset
      invariant ¬flag
      stop when now = lastmain ∨ now = lastset
      evolve    d(now) = 1
    trajdef postset
      invariant flag
      stop when now = lastmain
      evolve    d(now) = 1
```

4.4 Resolving Nondeterminism

There are three main sources of nondeterminism in TIOA: (1) the scheduling of actions (2) scheduling of trajectories, and (3) nondeterministic choices involving **choose** statements, **choose** parameters and **choose** expressions in initial assignments.

The nondeterminism resolution approach adopted by the TIOA simulator is to assign a program, called an *NDR program*, to each source of nondeterminism in an automaton. To aid the simulator in resolving nondeterminism a user is required to augment the automaton specification with a **schedule** block and **det** blocks each of which embodies an NDR program. A program in a **schedule** or a **det** block is used respectively for resolving automaton transitions, specifying the type and duration of trajectories and for resolving the values of a **choose** statement.

The NDR language for TIOA-sim is formally specified in Appendix C.2. Informally, an NDR program consists of loop (**while**) statements, conditionals (**if**), assignments, as well as statements that allow firing discrete transitions (**fire**) and continuous trajectories (**follow**).

Example 4.2 The following is a sample schedule block for simulating `TwoTaskrace` from Example 4.1. It chooses a time period at random from an interval determined by the minimum of the first variables and the maximum of the last variables. The **follow** `trajectory` statements cause the simulator to follow a trajectory defined by the trajectory definition `preset` if `flag` equals `false` and `postset` otherwise, for a time period defined by `passTime`. After allowing time to pass, an enabled action is chosen and fired. This sequence of events is iterated until the simulator reaches the maximum number of steps it is instructed to run for.

```
schedule
  states
    passTime: Real,  first: Real,  last: Real
  do
    while true do
      first := min(firstmain,firstset);  last := min(lastmain, lastset);
      passTime := now - randomReal(first,last);
      if ¬flag then
         follow trajectory preset for passTime
      else
         follow trajectory postset for passTime
      fi;
      if ¬flag ∧ now ≥ firstmain then
        fire internal increment
      elseif  ¬flag ∧ now ≥ firstset then
        fire internal set
      elseif  flag ∧ count > 0 ∧ now ≥ firstmain  then
        fire internal decrement
      elseif  flag ∧ count = 0 ∧ reported ∧ now ≥ firstmain then
        fire output report
      fi
    od
  od
```

4.5 Paired simulations

The TIOA simulator offers paired simulations to help users check whether an automaton A implements an automaton B that is typically specified at a higher level of abstraction. We refer to A and B as the low-level automaton and the high-level automaton respectively. Users present the paired simulator with descriptions of two automata, a candidate simulation relation, and a mapping, called a *step correspondence*, from the actions of the lower-level automaton A to sequences of actions of the higher-level automaton B, and from trajectories of A to sequences of trajectories of B. The simulator runs the low-level automaton A, checks whether the trace of the high-level automaton B induced by the step correspondence is identical to that of A, and checks whether the candidate simulation relation holds throughout the simulated executions.

A step correspondence needs to specify, for a given low-level transition or trajectory, a high-level execution fragment with the same trace such that the simulation relation holds between the respective final states of the transition or trajectory and the execution fragment. Thus, a step correspondence can be seen as an "attempted proof" of the simulation relation, missing only the reasoning that shows that the simulation relation is preserved. The NDR extensions to TIOA-sim (see Appendix C.3) allow the use of **proof** blocks to specify the proposed proof of a simulation relation. A **proof** block contains one entry for each possible transition definition in the low-level automaton, and one entry for each possible trajectory in the low-level automaton. Each entry encodes

an algorithm for producing a high-level execution fragment, using a program similar to the NDR programs used in automaton **schedule** blocks. In addition to these entries, the **proof** block also contains a **start** section, which specifies how to set the variables of the high level automaton given the initial state of the low-level automaton, and an optional **states** section that declares auxiliary variables used by the step correspondence.

Example 4.3 Suppose that we want to prove lower and upper time bounds for the occurence of a `report` action of `TwoTaskRace` automaton from Example 4.1. A common method for doing such a proof is to define a new automaton that specifies the property that the report action occurs within the time bounds we want to prove and then exhibit a simulation relation from `TwoTaskRace` to this new automaton. This new automaton is typically specified at a higher-level of abstraction than `TwoTaskRace` such that it just captures the time bounds. The automaton `TwoTaskRaceSpec` given below is a high-level automaton that we can use in our proof.

```
automaton TwoTaskRaceSpec(a1,a2,b1,b2: Real)
  signature
    output report
  states
    reported: Bool := false,
    now: Real := 0,
    firstreport: discrete Real :=  a1, % assuming a2 ≥ b1
    lastreport: discrete AugmentedReal := b2 + a2 + b2 * a2 / a1
  transitions
    output report
      pre ¬reported ∧ now ≥ firstreport
      eff reported := true;
          firstreport := 0;
          lastreport := infty
  trajectories
    trajdef prereport
      invariant ¬reported
      stop when now = lastreport
      evolve    d(now) = 1
    trajdef postreport
      invariant  reported
      evolve     d(now) = 1
```

We can use the paired simulation feature of the TIOA simulator to increase our confidence that `TwoTaskRace` implements `TwoTaskRaceSpec` before attempting to do the proof. In particular, we can do this by checking for a specified finite number of steps the validity of a candidate forward simulation relation from `TwoTaskRace` to `TwoTaskRaceSpec`.

4.5.1 Simulation relations

A simulation relation in TIOA is a predicate on the state variables of two automata.

Example 4.4 In the code below, we specify a forward simulation from the implementation automaton (TTR) to the specification one (TTRSpec).

```
forward simulation from TTR to TTRSpec:

  TTRSpec.now = TTR.now ∧
  TTRSpec.reported = TTR.reported ∧
  ¬TTR.flag ∧ (TTR.lastmain < TTR.firstset) ⇒
    TTRSpec.firstreport ≤
```

```
    min(TTR.firstset, TTR.firstmain) + TTR.count +
      ((TTR.firstset - TTR.lastmain) / a2) * a1 ∧
TTR.flag ∨ (TTR.lastmain ≥ TTR.firstset) ⇒
  TTRSpec.firstreport ≤ TTR.firstmain + TTR.count * a1 ∧
¬TTR.flag ∧ (TTR.firstmain ≤ TTR.lastset) ⇒
  TTRSpec.lastreport ≥ TTR.lastset +
    (TTR.count + 2 + (TTR.lastset - TTR.firstmain)/a1)*a2 ∧
(¬TTR.reported ∧ (TTR.flag ∨ (TTR.firstmain > TTR.lastset))) ⇒
  TTRSpec.lastreport ≥ TTR.lastmain + TTR.count * a2
```

4.5.2 Proof blocks

The language of proof blocks in TIOA can be used to specify, for each **fire** and **follow** statements in the low-level automaton, a corresponding execution fragment in the high-level automaton. A **fire** statement can be matched by a sequence of **fire** statements, while a **follow** statement can be matched by a sequence of statements containing both **follow** statements and **fire** statements of actions of type **internal**.

Example 4.5 The proof block below specifies a step correspondence for the `report` action as well as the `preset` and `postset` trajectories.

```
proof
  for output report do
    fire output report
  od

  for trajectory preset duration x do
    if (reported)
      follow pre report duration x
    else
      follow post report duration x
    fi
  od

  for trajectory postset duration x do
    if (reported)
      follow pre report duration x
    else
      follow post report duration x
    fi
  od
```

4.5.3 User interface and visualization

We have performed exploratory work on the TIOA simulator to enhance the usability of the tools by providing an effective graphical user interface and the ability to visualize simulation.

The TIOA simulator provides a programmatic interface for both a Text User Interface and a Graphical User Interface. In Phase I we fully implemented the Text User Interface and we experimented with a Graphical User Interface. We prototyped a connection of the simulator to the Eclipse integrated development environment. This proof-of-concept implementation enables the user to run the simulator as an "external tool" on an open IL (Intermediate Language) file produced by the front end. The output will appear in the Eclipse console. A preview of the this interface through the Eclipse platform is shown in Figure 4.5.3

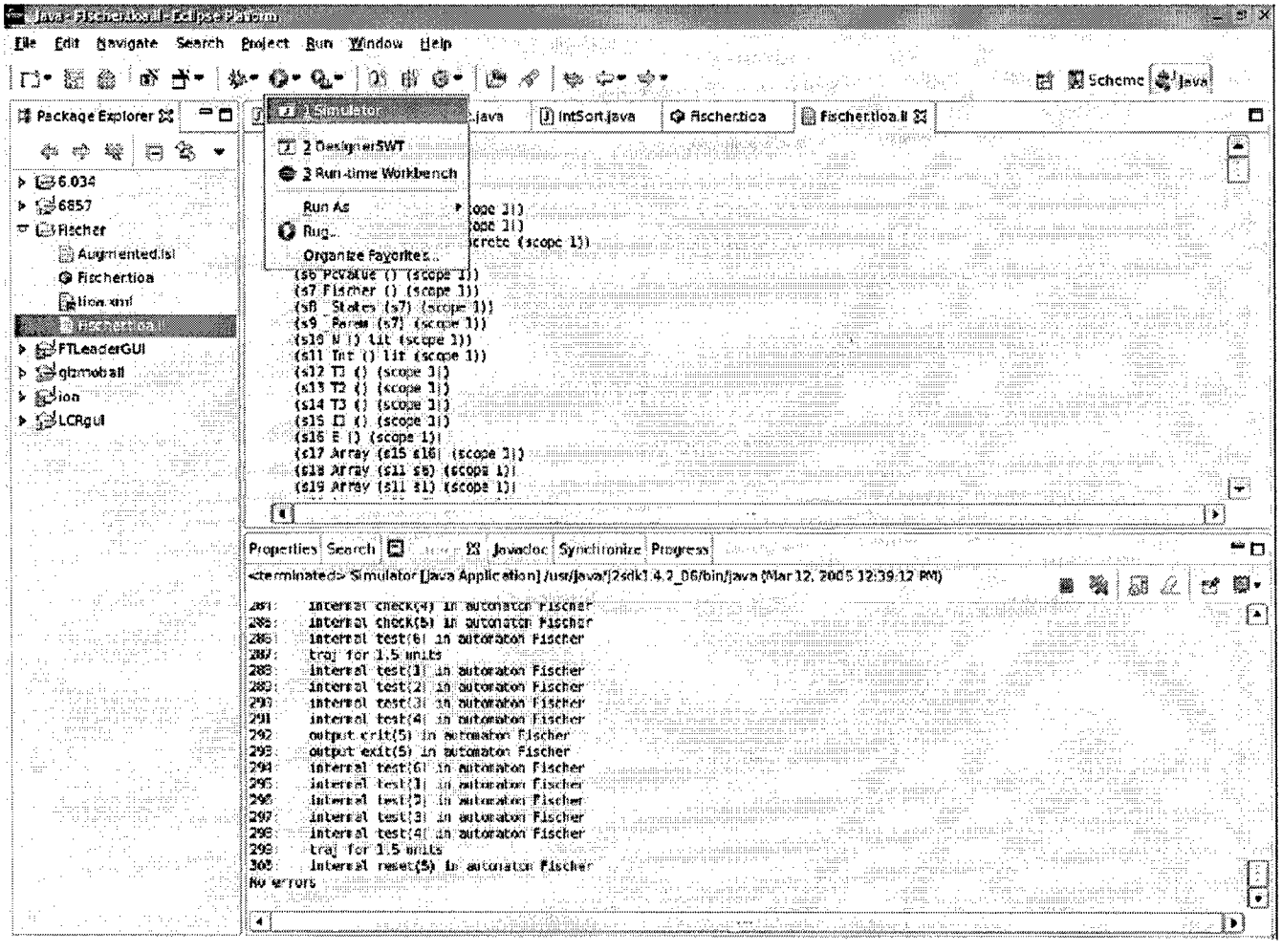

Figure 7: The TIOA Simulator User Interface

Textual user interface The simulator outputs to the console the trace of the automaton's simulation, that is, the transitions fired, trajectories followed, and optionally the state variables that change. Other events such as initialization, errors (such as firing transitions whose preconditions do not hold, or trajectories whose invariants do not hold or stop conditions have passed) are also output to the console.

Graphical user interface We prototyped a GUI for the simulator, the TIOA Visualizer. We used the visualizer to view a simulation of the LCR Leader Election Algorithm on a number of nodes connected in a ring structure. The Visualizer displays each node and the channels that connect the nodes as well. When an action occurs, the visualizer acts accordingly. In our prototype, the reaction of the visualizer was hard-coded in the GUI; when a SEND(message, i, j) action occurs, for example, *message* can be seen leaving node i and being stored in the buffer of the channel connecting nodes i and j. When a leader is elected, the node changes color. A view of the tool is shown in Figure 8.

5 TIOA and Theorem-Proving Tools

The TIOA model provides a precise and expressive linguistic framework for developing specifications for a broad class of systems. Earlier experiences [57, 66, 84] with timed and hybrid automata indicate that inductive verifi-

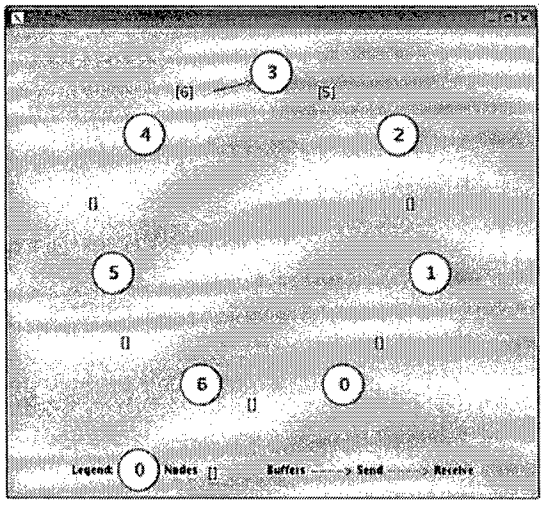

Figure 8: LCR Leader Election Visualization

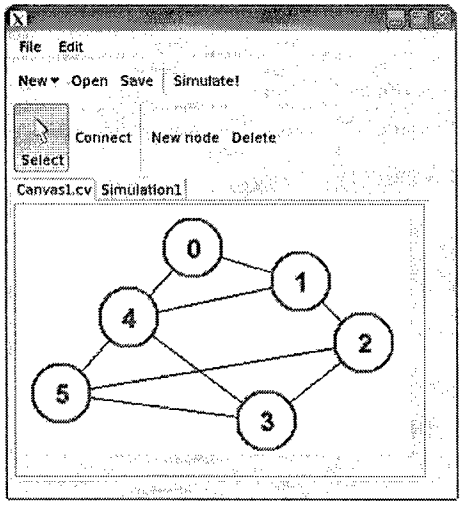

Figure 9: A graphical tool for designing visualizations

cation of moderately complex systems calls for considerable time and effort. A theorem prover can significantly reduce the human effort involved in the verification process, and it can also improve the quality of the proofs by (1) automatically resolving many of the simpler proof goals, (2) allowing proof reuse, and (3) managing and organizing large unwieldy proofs. In Phase I we developed a *TIOA interface* for the Prototype Verification System (PVS) [87] developed at SRI. PVS is a general-purpose interactive theorem prover for high order logic, and it has been used to verify a variety of systems, both from the industry and academia [91]. The motivation for developing a TIOA-specific interface for PVS is two fold. First, it will relieve the TIOA users from having to master PVS's own specification language and its quirks. Second, with a suite of TIOA-specific PVS proof strategies, it will be possible to limit the human interaction to the essential parts, and also to generate human-readable proofs.

5.1 Overview

The key technique used for the analysis of TIOA models is deduction. It is therefore desirable to be able to reason about timed I/O automata using interactive theorem provers because: (1) modern theorem provers can efficiently manage large proofs, and (2) it is possible to automate proofs of recurring proof patterns by writing theorem-prover macros or strategies. A cost one has to bear for using a theorem prover is that of writing the description of the TIOA model of the system in yet another language, i.e., the language of the prover, which is typically some variant of higher order logic. In Phase I of the project, we prototyped a tool that automatically translates system descriptions written in the TIOA language to PVS theories. We have used the translator, called `tioa2pvs`, in three verification case studies to produce PVS theories from TIOA descriptions, and have successfully used PVS to prove interesting properties of the systems under study.

Written in Java, the translator builds upon the existing IOA to Larch translator [8]. It first uses the TIOA front-end type checker to parse the input TIOA, reporting any errors if necessary (see Figure 10). The translator then generates a set of files containing PVS theories specifying the automata and their properties. The user can then invoke the PVS theorem prover to interactively prove these properties.

5.2 Translation process

Our approach for translating TIOA to PVS is based on the methodology described in the Timed Automata Modeling Environment (TAME) [2, 3]. The translator automatically instantiates the TAME template with the states, actions and transitions of the input timed I/O automaton written in TIOA. This instantiated theory, together with several supporting library theories, completely specifies the automaton, its transitions and its reachable states in

the language of PVS.

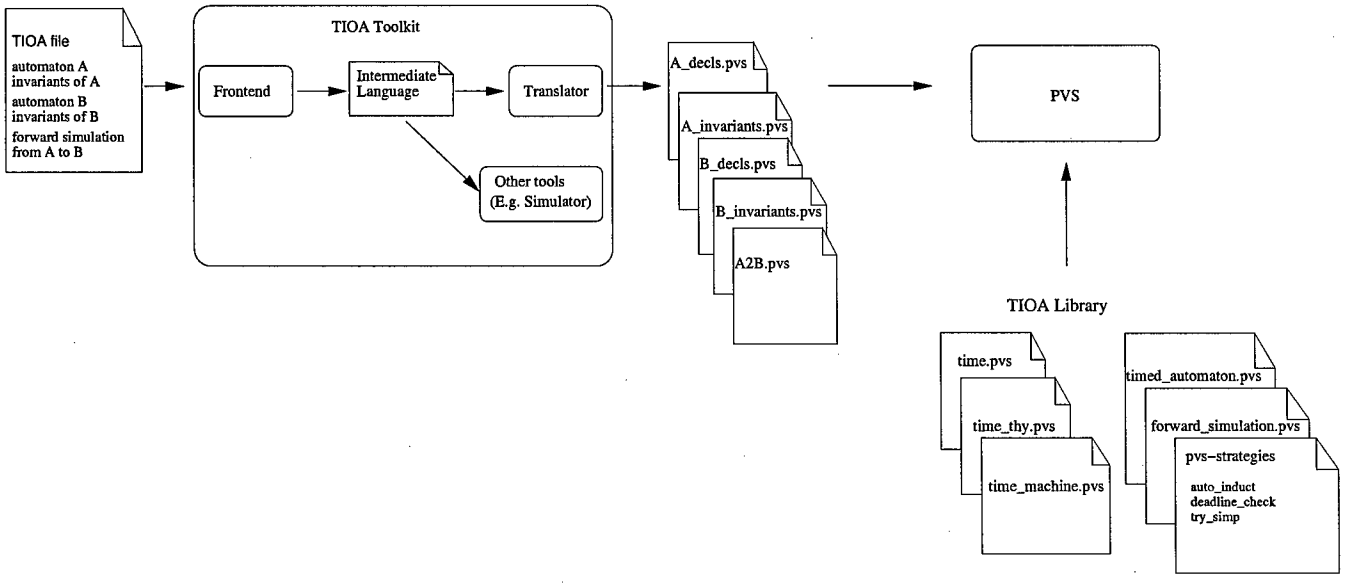

Figure 10: Theorem proving on TIOA specifications

There is an important distinction between the automaton model for which TAME was designed and the timed I/O automaton model. Elapsing of time in a TAME automaton is captured by a special time-passage action that increments a special `now` variable, whereas in the more general timed I/O automaton model, the executions contain trajectories that map intervals of time to values of continuously changing variables. To translate these trajectory definitions to PVS, we add a set of parameterized actions that contain the trajectory map as a parameter. The precondition of these actions enforces the trajectory invariants and the stopping conditions, while the effect of these actions returns the last state of a trajectory.

The TIOA language allows explicit nondeterminism within transitions of actions using the **choose** statement, and within trajectories as differential inclusions. We convert this form of explicit nondeterminism in the TIOA language into implicit nondeterminism in PVS by introducing additional action parameters for the variables with nondeterministic values. The precondition of these actions requires that these additional parameters have values allowed by the nondeterministic choice specified in the TIOA description.

We next describe in more detail the translation of the various components of a TIOA description.

5.2.1 Data types

Primitive data types of the TIOA language (e.g. Bool, Char, Int, Nat, Real, String) have their equivalents in PVS, which also supports declaration of enumeration, unions, arrays and tuples in its own syntax. The TIOA language introduces the type `AugmentedReal`, which is the type `Real` extended with a constructor for infinity. `AugmentedReal` is translated to the PVS type `time` introduced in the time theory of TAME.

The TIOA language allows use of user-defined types and operators by requiring the user to declare the types and the signature of the operators within the TIOA description. This feature allows the user to use existing PVS theories and data structures without having to rewrite them in TIOA. The user can also use command-line flags to specify the location of the files containing the PVS definitions, so that the output of the translator imports these definitions accordingly. `Timedqueue` is an example of a user-defined type, which is used in the failure-detection case study. The TIOA description declares this type and its operators, like `addqueue`, `delqueue`, and uses this type and these operators in the automaton description. The semantics of these user-defined types and operators are written as a PVS library theory along with the required properties of `addqueue` and `delqueue`.

The translator associates the TIOA types with the corresponding library theory (if there exists one) by importing the relevant theory in the output.

5.2.2 States, actions and transitions

The TIOA language provides the explicit constructs for specifying the state variables, actions and transitions of an automaton.

PVS state: In PVS, the state component of an automaton is defined as a record containing the various state variables. A boolean predicate `start` returns true when a given state satisfies the conditions of a start state.

Actions and preconditions: Individual actions are declared as subtypes of an `action` data type in PVS. Preconditions of actions are translated as a combined parameterized predicate on a given action and state. This predicate returns true when the given action is enabled at the given state.

Transitions: All imperative statements in the effects of transition definitions, including assignment statements and conditionals, are replaced by a function that returns the poststate when given an action and a prestate as parameters.

5.2.3 Trajectories

To translate trajectory definitions to PVS, we add a set of parameterized actions that contain the trajectory map as a parameter. The precondition of these actions enforces the trajectory invariants, the stopping conditions, and the evolution of time-valued variables. The effect of these actions returns the last state of a trajectory by applying the trajectory map to a given time interval. The current implementation of the translator handles only constant differential equations and inclusions of the form $d(x) = k$ or $k_1 \leq d(x) \leq k_2$.

5.2.4 Invariants and simulation relations

The translator supports translation of invariants and simulation relations written in a TIOA description. Separate PVS theory files are generated for the automaton specifications, the invariants and the simulation relations. An invariant in TIOA is translated to a lemma in PVS specifying that if a state is reachable, then the condition of the invariant holds in the state.

For describing simulation relations in PVS, TAME provides an automaton theory template and a forward simulation theory template. The automaton theory template specifies the generic structure of an automaton. The forward theory simulation template accepts the following as parameters: two instances of the automaton theory template, a function mapping actions of the first automaton to actions of the second automaton, and a predicate describing the relation among the state variables of the two automata. The forward simulation theory template defines a forward simulation from one automaton to the other in terms of these input parameters. Our translator outputs the desired simulation relation by instantiating this forward simulation theory template with the theory interpretations of the abstract and concrete automata, an action map and the relation among the state variables of the two automata.

5.2.5 Auxiliary functions

The TIOA language allows function declarations as a means of convenience and usability. The user can declare a derived variable whose value is a function of other state variables, or a macro-style function for shorthand, or a function involving mathematical expressions. Functions can be declared at the top level of a TIOA description, or within various components of an automaton.

Top level auxiliary functions are translated into top level functions in PVS with minimal syntax modifications. In a TIOA description, auxiliary functions declared within an automaton can be used in the preconditions and transitions of actions as well as in trajectories. In PVS, the preconditions are translated as a predicate parameterized on an action and a state, separately from the transitions, which are translated as functions that return

the poststate given by an action and a prestate as parameters. The translator outputs a function declared within an automaton as a top level function in PVS so that it can be used in both the precondition and transition components in PVS. In PVS, these top level functions have additional parameters for accessing the state variables and the action parameters.

5.2.6 Auxiliary rewrite rules

The translator automatically generates PVS theories containing auxiliary lemmas that can be used as rewrite rules. These auxiliary rewrite rules can be used in proofs of other lemmas. The proof obligations of these auxiliary rewrite rules are easily discharged using TAME strategies. The translator also automatically generates the proof scripts for these lemmas. The user just has to run the generated proof scripts in PVS to complete these proof obligations.

5.3 Translation case studies

We have used the translator successfully in the following three verification case studies: (1) the Fischer mutual exclusion algorithm, (2) a two-task race system, and (3) a simple failure detector. In all these case studies, the input to the translator is a TIOA description of the system/algorithm in question and its invariant properties. For 2 and 3 we also have an abstract TIOA specification of the timing properties of the system. The output from the translator is a set of PVS theories specifying the timed I/O automata and their invariant properties. We have used the PVS theorem prover to verify the properties using inductive invariant proofs and simulation relations.

See Section 6 and the appendices for selected information and specifications.

5.4 Use of strategies in PVS

In the proofs of invariants in the case studies, we used high-level PVS strategies developed for TAME. These strategies are similar to ones we developed in earlier case studies for untimed automata, in which we used the Larch Prover instead of PVS. The strategy `auto_induct` is used in invariant proofs at the top level for applying induction over the number of steps of execution. Resulting branches correspond to the base case of the induction and the case splits for various actions in the induction step. Using this strategy, trivial branches are immediately discharged. Within a branch corresponding to an action step, the user can invoke `apply_specific_precond` to obtain the precondition assertion for use in the proof. The `try_simp` strategy attempts to discharge straightforward proof obligations. For branches involving time passage actions corresponding to TIOA trajectories, the strategy `deadline_check` examines the stopping condition of the trajectory and provides an appropriate time interval for instantiating a trajectory map.

6 Case Studies and Usage Patterns

In Phase I we developed several examples that illustrate and test the use of the prototype tools.

Fischer mutual exclusion example

We developed a TIOA specification for the Fischer mutual exclusion (mutex) algorithm (as given in [42]). The specification was processed and type-checked by the front end. The front end generated an intermediate output file that was used by the `tioa2pvs` translator to generate the PVS specifications that were then used to prove the mutual exclusion property. The complete presentation and the listings of the files is given in Appendix A. We have also simulated executions of the TIOA specification.

TwoTaskRace example

In the two-task race system, we studied a simple timed specification where one automaton increments a counter until it is interrupted by an action setting the counter. Thereafter, the automaton decrements the counter and reports when it reaches 0. To prove that the report occurs within a certain time bound, we create an abstract automaton that simply performs a report action within the required time bounds. We then prove a forward simulation relation from the concrete automaton to the abstract automaton, thus showing that the timing requirements are met. The detailes of this example can be found in Appendix B.

Verification of the SATS

We have also considered a model for the Small Aircraft Transportation System (SATS), a system for landing aircraft in small and medium airports currently being studied as a joint project of NASA and several universities. Our study on this model is based on the work of Cesar Munoz in National Institute of Aerospace (NIA), and his colleagues [19], on which a mathematical model for the system was presented. They modeled the system by discretizing the space of the airport into several zones represented as queues of aircraft, and under such a discrete system model, some interesting properties of the system are proved by a state exploration method using a mechanical theorem prover PVS. These properties include the bounds on the number of aircraft in a specific zone of the system and the number of simultaneous operations of the aircrafts.

One objective for us to conduct research on this model is to see if we can prove the properties presented in the paper by means of a standard invariant-proof technique using tools, techniques, and experiences we have used from earlier examples. We chose some properties from the original work and developed a revised PVS specification from the original model (revised so that it becomes more handy for invariant-style proofs, without the changes in the semantics.)

The difficulties are in having auxiliary lemmas to make the inductive proofs go through, as opposed to the state exploration method, which will explore the entire state spaces to directly check the properties. We developed such lemmas and now working on the proof of them in PVS.

The experience gained in this case study will be used in Phase II for developing proof strategies that can be used to help automate the theorem-proving process. We have also been able to use the translator to translate the original TIOA specification to PVS and we believe this resulting specification can be verified with greater ease once the strategies are in place.

7 Conclusions and Future Work

We have completed Phase I project having demonstrated the feasibility of implementing a framework for modeling and analyzing complex distributed systems. We developed a detailed design of a modeling language, TIOA, that includes event ordering behavior and timing behavior. We developed an implementation of a front end for this language, together with prototype tools to support simulation (including simulation at multiple levels of abstraction). We developed a translator from TIOA to PVS and demonstrated the feasibility of prooding invariants and abstraction relationships of TIOA-specified systems using PVS. We developed examples that demonstrate the use of the tools. We explored ease-of-use features in the framework that include a graphical user interface and an approach to user-defined visualizations.

We are now poised to transition the project to Phase II in which we will complete this development and implement production-grade, well-integrated computer-aided tools that together will comprise a user-oriented framework for modeling and analyzing complex distributed systems. Our proposed Phase II work for the next two years is overviewed in Appendix D.

References

[1] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Myla M. Archer, Constance L. Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In *Workshop on User Interfaces for Theorem Provers*, Eindhoven University of Technology, July 1998.

[3] Myla Archer and Constance Heitmeyer and Elvinia Riccobene, Proving Invariants of I/O Automata with TAME. *Automated Software Engineering*, 9(3), pages 201–232, 2002

[4] Myla Archer. TAME: PVS Strategies for special purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1/4), February 2001.

[5] M. Cecilia Bastarrica, Rodrigo E. Caballero, Steven A. Demurjian, Alexander A. Shvartsman: Two Optimization Techniques for Component-Based Systems Deployment. SEKE 2001: 153-162, 2000

[6] M. Cecilia Bastarrica, Steven A. Demurjian, Alexander A. Shvartsman: Comprehensive Specification of Distributed Systems Using I5 and IOA. SCCC 2000: 74-82, 1999

[7] M. Cecilia Bastarrica, Steven A. Demurjian, Alexander A. Shvartsman: A Framework for Architectural Specification of Distributed Object Systems. OPODIS 1999: 127-148, 1999

[8] Andrej Bogdanov, Stephen J. Garland, and Nancy A. Lynch. "Mechanical translation of I/O automata specifications into first-order logic," Formal Techniques for Networked and Distributed Systems, FORTE 2002, 2002, LNCS 2529, Doron A. Peled and Moshe Y. Vardi (eds.), Springer-Verlag, pages 364-368.

[9] A. Bouajjani and S. Tripakis and S. Yovine. On-the-fly Symbolic Model checking for Real-time Systems, 18th IEEE Real-Time Systems Symposium (RTSS'97), San Francisco, CA, IEEE, pages 25–34, December, 1997

[10] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Co., Reading, MA, 1988.

[11] Oleg Cheiner. Implementation and evaluation of an eventually-serializable data service. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1997.

[12] Oleg Cheiner and Alex Shvartsman. Implementing an eventually-serializable data service as a distributed system building block. In M. Mavronicolas, M. Merritt, and N. Shavit, editors, *Networks in Distributed Computing*, volume 45 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–72. American Mathematical Society, 1999.

[13] B.S. Chlebus, R. De Prisco, and A.A. Shvartsman. Performing tasks on synchronous restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.

[14] E.M. Clarke and E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, Proc. Workshop on Logic of Programs, pages 52–71", LNCS 131, Springer, 1981

[15] C. Daws and A. Olivero and S. Tripakis and S. Yovine. The tool Kronos, Hybrid Systems III, Verification and Control, Springer-Verlag, pages 208–219, LNCS, volume 1066, 1996

[16] Roberto De Prisco, Alan Fekete, Nancy Lynch, and Alex Shvartsman. A dynamic view-oriented group communication service. In *Proceedings of the 17th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 227–236, Puerto Vallarta, Mexico, June-July 1998.

[17] R. De Prisco, A. Fekete, N. Lynch, and A.A. Shvartsman. A dynamic primary configuration group communication service. *Distributed Computing* Proceedings of DISC'99 - 13th International Symposium on Distributed Computing, Bratislava, Slovak Republic, September 1999, volume 1693 of *Lecture Notes in Computer Science*, pages 64–78, Bratislava, Slovak Republic, 1999. Springer-Verlag-Heidelberg.

[18] RFC 3315 - Dynamic Host Configuration Protocol for IPv6 (DHCPv6), R. Droms (Ed.), July 2003

[19] Gilles Dowek, Cesar Munoz and Victor A. Carreno, Abstract Model of the SATS Concept of Operations: Initial Results and Recommendations , NASA/TM-2004-213006, March 2004, pp. 46

[20] Rui Fan and Nancy Lynch. Efficient replication of large data objects. In *DISC 2003: 17th International Symposium on Distributed Computing*, Sorrento, Italy, October 2003. To appear.

[21] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data service. *Theoretical Computer Science*, 220(1):113–156, June 1999. Special Issue on Distributed Algorithms.

[22] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, May 2001.

[23] Stanislav Funiak, Model checking IOA programs with TLC, MIT Laboratory for Computer Science, July 2001. http://theory.lcs.mit.edu/tds/papers/Funiak

[24] S. Garland, N. Lynch, Joshua Tauber, and M. Vaziri. *IOA User Guide and Reference Manual.* MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2003. Available at `http://theory.lcs.mit.edu/tds/ioa.html`.

[25] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[26] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, USA, 2000.

[27] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A language for specifying, programming and validating distributed systems, October 2001. User and Reference Manual. `http://theory.lcs.mit.edu/tds/ioa-manual.ps`.

[28] Stephen J. Garland and Nancy A. Lynch, Model-Based Software Design and Validation, United States Patent No. 6,289,502 September 11, 2001.

[29] Ch. Georgiou, P. Musial, and A. Shvartsman, Long-Lived Rambo: Trading Knowledge for Communication, Proc. of 11th International Colloquium on Structure of Information and Communication Complexity (SIROCCO04), to appear, LNCS, 2004

[30] Ch. Georgiou, A. Russell and A. Shvartsman. The Complexity of Synchronous Iterative Do-All with Crashes. *Distributed Computing*, Volume 17, No 1, pages 47-63, 2004.

[31] Ch. Georgiou, A. Russell and A. Shvartsman. Work-Competitive Scheduling for Cooperative Computing with Dynamic Groups, *The 35th Annual ACM Symposium on Theory of Computing (STOC'2003)*, 2003.

[32] Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *International Conference on Dependable Systems and Networks*, pages 259–268, San Francisco, CA, June 2003.

[33] Williem Otto David Grifioen, Studies in Computer Aided Verification of Protocols, Doctoral Dissertation, Catholic University of Nijmegen, May 2000.

[34] R. Grosu and S. A. Smolka, Monte Carlo Model Checking, Proceedings of TACAS 2005, Springer-Verlag, 2005

[35] Vida Uyen Ha. Verification of an attitude control system, May 2003. Bachelor of Science and Master of Engineering.

[36] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994. IEEE.

[37] Constance Heitmeyer and Nancy Lynch. Formal verification of real-time systems using timed automata. In Constance Heitmeyer and Dino Mandrioli, editors, *Formal Methods for Real-Time Computing*, Trends in Software, chapter 4, pages 83–106. John Wiley & Sons Ltd, April 1996.

[38] Tobias Hamberger, Integrating theorem proving and model checking in Isabelle/IOA, Technical Report, Technische Universitat Munchen, 1999.

[39] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for Ensemble layers. In Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Fifth International Conference, (TACAS'99), 1999*, LNCS 1579, pages 119–133. Springer-Verlag, 1999.

[40] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. The theory of timed I/O automata. Technical Report MIT/LCS/TR-917, MIT Laboratory for Computer Science, 2003. Available at `http://theory.lcs.mit.edu/tds/reflist.html`.

[41] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. Technical report, Cancun, Mexico, 2003. Full version available as Technical Report MIT/LCS/TR-917.

[42] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *Theory of timed I/O automata.* Technical Report MIT-LCS-TR-917, MIT CSAIL Cambridge, MA 02139. Monograph to appear in Synthesis Series, Morgan-Claypool publishers, 2005

[43] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. A framework for modeling timed systems with restricted hybrid automata. In *RTSS 2003: The 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, December 2003.

[44] Dilsun Kirli Kaynar, Anna Chefter amd Laura Dean, Stephen Garland, Nancy Lynch, Toh Ne Win, and Antonio Ramírez-Robredo. The ioa simulator, 2002. Manuscript available from http://theory.lcs.mit.edu/ dilsun/Publications.html.

[45] Dilsun Kirli Kaynar, Anna Chefter, Laura Dean, Stephen J. Garland, Nancy A. Lynch, Toh Ne Win, and Antonio Ramírez-Robredo. Simulating nondeterministic systems at multiple levels of abstraction. In *Proceedings of Tools Day affiliated to CONCUR 2002*, Brno, Czech Republic, August 2002.

[46] I. Keidar and R. Khazan. A virtually synchronous group multicast algorithm for WANs: Formal approach. *SIAM Journal on Computing*, 32(1):78–130, 2003.

[47] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20thIEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 344–355, Taipei, Taiwan, April 2000.

[48] Idit Keidar, Roger Khazan, Nancy Lynch, and Alex Shvartsman. An inheritance-based technique for building simulation proofs incrementally. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):1–29, January 2002. Conference version in *ICSE 2000*, pp. 478–487.

[49] Robust Collaborative Multicast Service for Airborne Command and Control Environment. Military Communications Conference (MILCOM), Monterey, CA, October 31-November 3, 2004. (Fred W. Ellersick MILCOM Award for Best Paper in the Unclassified Technical Program, 2004)

[50] Roger Khazan, Alan Fekete, and Nancy Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on Distributed Computing*, pages 258–272, Andros, Greece, September 1998.

[51] Roger Khazan and Nancy Lynch. An algorithm for an intermittently atomic data service based on group communication. DISC'03 Florence, Italy, October 2003.

[52] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[53] Leslie Lamport, Specifying concurrent systems with TLA+, Calculational System Design, M. Broy and R. Steinbruggen, editors, IOS Press, Amsterdam, 1999, pages 183–247.

[54] Butler Lampson, Nancy Lynch, and Jørgen Søgaard-Andersen. At-most-once message delivery: A case study in algorithm verification. In W. R. Cleaveland, editor, *CONCUR'92* (Third International Conference on Concurrency Theory), volume 630 of *LNCS* , pages 317–324. Springer-Verlag, 1992.

[55] K. Larsen and P. Petterson and W. Yi, Uppaal in a nutshell, Software Tools for Technology Transfer, volume 1, number 1/2, October, 1997

[56] Gunter Leeb and Nancy Lynch. Proving safety properties of the steam boiler controller: Formal methods for industrial applications: A case study, 1996. *Lecture Notes in Computer Science*, Springer-Verlag.

[57] Carl Livadas, John Lygeros, and Nancy Lynch. High-level modeling and analysis of TCAS. In *Proceedings of the 20th Real-Time Systems Symposium (RTSS99)*, Phoenix, Arizona, December 1999. `theory.lcs.mit.edu/tds/papers/Livadas/RTSS99.ps.gz`.

[58] Carolos Livadas, Formally Modeling, Analyzing, and Designing Network Protocols — A Case Study on Retransmission-Based Reliable Multicast Protocols, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, July, 2003

[59] Carolos Livadas and Nancy Lynch. A reliable broadcast scheme for sensor networks. Technical Report 915, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, August 2003.

[60] V. Luchangco, E. Söylemez, S. Garland, and N. Lynch. Verifying timing properties of concurrent algorithms. In D. Hogrefe and S. Leue, eds., *Formal Description Techniques VII: Proc. of 7th IFIP WG6.1 Int-l Conf. on Formal Descr. Techniques* (FORTE'94), pp. 259–273. Chapman & Hall, 1995.

[61] John Lygeros and Nancy Lynch. On the formal verification of the TCAS conflict resolution algorithms. In *36th IEEE Conference on Decision and Control*, pages 1829–1834, San Diego, CA, December 1997.

[62] John Lygeros and Nancy Lynch. Strings of vehicles: Modeling and safety conditions. In S. Sastry and T.A. Henzinger, editors, *Hybrid Systems: Computation and Control* (First International Workshop, HSCC 1998), LNCS 1386 pages 273–288. Springer Verlag, 1998.

[63] John Lygeros and Nancy Lynch. Conditions for safe deceleration of strings of vehicles. Research Report UCB-ITS-PRR-2000-2, California PATH Program, Institute of Transportation Studies, University of California, Berkley, January 2000.

[64] Nancy Lynch. Modelling and verification of automated transit systems, using timed automata, invariants and simulations. Technical Memo MIT/LCS/TM-545, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, December 1995. Also, [66].

[65] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[66] Nancy Lynch. Modelling and verification of automated transit systems, using timed automata, invariants and simulations. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III: Verification and Control* (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, 1995), LNCS 1066 pages 449–463. Springer-Verlag, 1996. Also, [64].

[67] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata revisited. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control. 4th Int-l Workshop (HSCC'01)*, LNCS 2034, pages 403–417. Springer-Verlag, 2001.

[68] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Compositionality for probabilistic automata. In *CONCUR 2003: 14th Int-l Conference on Concurrency Theory*, Marseille, France, September 2003.

[69] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Compositionality for probabilistic automata. In Roberto Amadio and Denis Lugiez, editors, *CONCUR 2003: Concurrency Theory (The 14th International Conference on Concurrency Theory, 2003)*, LNCS 2761 pages 208–221. Springer, 2003.

[70] Nancy Lynch, Nir Shavit, Alex Shvartsman, and Dan Touitou. Timing conditions for linearizability in uniform counting networks. *Theoretical Computer Science*, 220(1):67–91, June 1999.

[71] N. Lynch and A. Shvartsman, *A Framework for Modeling and Analyzing Complex Distributed Systems*, Proposal to AFOSR, Topic No. AF04-T023, Proposal No. F045-023-0117, VEROMODO, Inc., April 15, 2004 (revised August 10, 2004).

[72] Nancy Lynch and Alex Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 272–281, Seattle, Washington, USA, June 1997. IEEE.

[73] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In D. Malkhi, editor, *Distributed Computing (Proceedings of the 16th International Symposium on DIStributed Computing (DISC), 2002)*, LNCS 2508 pages 173–190. Springer-Verlag, 2002.

[74] Nancy Lynch and Frits Vaandrager. Forward and backward simulations — Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[75] Nancy Lynch and H. B. Weinberg. Proving correctness of a vehicle maneuver: Deceleration. In *Second European Workshop on Real-Time and Hybrid Systems*, pages 196–203, Grenoble, France, May/June 1995. Later version appears as [107].

[76] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[77] Nancy A. Lynch and Frits W. Vaandrager. Action transducers and timed automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[78] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, LNCS 600, pages 447–484, 1992.

[79] Catherine Matlon, "A Specification and Verification of Intermittent Global Order Broadcast", MIT EECS M.Eng. thesis, June 2004.

[80] Michael Merritt, Francemary Modugno, and Mark R. Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, editors, *CONCUR'91: 2nd International Conference on Concurrency Theory,* volume 527 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1991.

[81] Sayan Mitra. HIOA+: Specification language and proof tools for hybrid systems, 2003. Work in progress, http://theory.lcs.mit.edu/ mitras/research/LCPTHIOA.ps.

[82] Sayan Mitra and Myla Archer. Developing strategies for specialized theorem proving about untimed, timed, and hybrid i/o automata. In *STRATA 2003*, Rome, Italy, 2003.

[83] Sayan Mitra, Yong Wang, Nancy Lynch, and Eric Feron. Application of Hybrid I/O automata in safety verification of pitch controller for model helicopter system. Technical Report MIT-LCS-TR-880, MIT Laboratory for Computer Science, Cambridge, MA 02139, January 2003.

[84] Sayan Mitra, Yong Wang, Nancy Lynch, and Eric Feron. Safety verification of model helicopter controller using Hybrid Input/Output automata. In O. Maler and A. Pnueli, editors, *Hybrid Systems: Computation and Control (6th Int-l Workshop, HSCC 2003)*, LNCS 2623, pages 343–358. Springer-Verlag, 2003.

[85] Cesar Munoz, Ricky W. Butler, Victor A. Carreo, and Gilles Dowek. On the Formal Verification of Conflict Detection Algorithms, NASA/TM-2001-210864, NASA Langley Research Center, May 2001.

[86] Atish Nigam, Enhancing the IOA code generator's abstract data types, MIT Laboratory for Computer Science, August 2001. http://theory.lcs.mit.edu/tds/papers/Nigam

[87] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, LNCS 1102, pages 411–414. Springer Verlag, 1996.

[88] T. P. Petrov, A. Pogosyants, S. J. Garland, V. Luchangco, and N. A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In R. Gotzhein and J. Bredereke, eds., *Formal Description Techniques IX: Theory, Applications, and Tools* (FORTE/PSTV'96: Joint Int-l Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, 1996), pages 29–44. Chapman & Hall, 1996.

[89] Lee Pike, Jeffrey Maddalon, Paul Miner, Alfons Geser. Abstractions for Fault-Tolerant Distributed System Verification. Theorem-Proving in Higher-Order Logics (TPHOLs), 2004.

[90] J. P. Queille and J. Sifakis, Specification and Verification of Concurrent Systems in Cesar, Proceedings of the International Symposium in Programming, Lecture Notes in Computer Science, Vol. 137, Springer-Verlag, Berlin, 1982

[91] http://www.csl.sri.com/users/rushby/pvs-bib.bib.txt

[92] J. Antonio Ramırez-Robredo. Paired simulation of I/O automata, September 2000. Master of Engineering and Bachelor of Science in Computer Science and Engineering Thesis, Massachusetts Institute of Technology, Cambridge, MA.

[93] Christine Margaret Robson. TIOA and UPPAAL. Masters of Engineering Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 2004

[94] John Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. In IEEE Transactions on Software Engineering, Volume 25, Number 5. September, 1999. Pages 651–660.

[95] R. Segala, R. Gawlick, J.F. Søgaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, March 1998.

[96] Roberto Segala, Rainer Gawlick, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, March 1998.

[97] Mark Smith. *Formal Verification of TCP and T/TCP*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1997.

[98] Jørgen Søgaard-Andersen. *Correctness of Protocols in Distributed Systems*. PhD thesis, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, December 1993. ID-TR: 1993-131. Also, expanded version in MIT/LCS/TR-589.

[99] Jørgen Søgaard-Andersen, Nancy A. Lynch, and Butler Lampson. Correctness of communication protocols: A case study. Technical Memo MIT/LCS/TR-589, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139, November 1993. Expanded version of [98].

[100] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification* (5th Int-l Conference, CAV'93, 1993), LNCS 697, pages 305–319. Springer-Verlag, 1993.

[101] Joshua A. Tauber. *Verifiable Code Generation from I/O Automata for Distributed Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 2004.

[102] Joshua A. Tauber and Nancy A. Lynch and Michael J. Tsai, Compiling IOA without Global Synchronization, Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA04), Cambridge, MA, 2004.

[103] Software Assurance, *MIT Technology Review*, 10 Emerging Technologies that will Change the World, Annual Innovation Issue, pp. 44–46, 2003.

[104] S. Tripakis and S. Yovine and A. Bouajjani, Checking Timed Büchi Automata Emptiness Efficiently, Formal Methods in System Design, Kluwer Academic Publishers, Accepted for publication

[105] Michael J. Tsai. Code generation for the IOA language, June 2002. Master of Engineering Thesis, Massachusetts Institute of Technology, Cambridge, MA.

[106] Christopher Walton, Dilsun K. Kaynar, and Stephen Gilmore. An abstract machine model of dynamic module replacement. In *Future Generation Computer Systems*, 16(7), pages 793-808, May 2000.

[107] H. B. Weinberg and Nancy Lynch. Correctness of vehicle control systems: A case study. In *17th IEEE Real-Time Systems Symposium*, pages 62–72, Washington, D. C., December 1996.

[108] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun K. Kaynar, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. In *Verification, Model Checking, and Abstract Interpretation (Proc. of 4th Int-l Conf., VMCAI 2003)*, LNCS 2575, pp. 283–297. Springer-Verlag, 2003.

[109] Yuan Yu, Panagiotis Maolios, and Leslie Lamport, Model checking TLA+ specifications, Correct Hardware Design and Verification Methods (CHARME '99), Laurence Pierre and Thomas Kropf, eds., LNCS 1703, September 1999, pages 54–66.

A Fischer Mutex Specifications

This appendix contains the four files for the Fischer Mutex examples.

1. **fischer_me.tioa**: The file used is TIOA specification of the mutex used as the input to TIOAtoPVS translator. It contains the statement of the invariants as well as the description of the automaton. The translator automatically outputs the pvs file for automata and invariants.
2. **fischer_me_decls.pvs**: The declaration of the automaton in PVS. It was automatically generated by the translator.
3. **fischer_me_invariants.pvs**: Invariants we want to prove in PVS. It was also automatically generated by the translator.
4. **common_decls.pvs**: This file was automatically generated by the translator. In this example, this file just imports other definitions needed in fischer_me_decls.pvs, for example, the definition of time. fischer_me_decls.pvs imports this file.

In this example fischer_me.tioa was type-checked, and went through translator. It generated three files, common_decls.pvs, fischer_me_decls.pvs, and fischer_me_invariants.pvs. All proofs for fischer_me_invariants.pvs were done in PVS.

Fischer Mutual Exclusion algorithm is an asynchronous single register mutex algorithm. We took the definition of the algorithm from example 4.5 in [42]. This composite version includes task bounds that are deadlines for the earliest and the latest time a transition can take place.

In more detail, the algorithm implements mutual exclusion through the use of timing constraints. Some of the transitions, such as try_i, $test_i$, $crit_i$, $exit_i$, $reset_i$ and rem_i just change the program counter and are not influenced by the timing constraints. However, once a process checks for the quiescent value of a shared turn variable $x = 0$ and enters a *set* state it assumes that it can proceed to the critical section. Imagine if several processes entered their *set* states simultaneously. Now all of the processes are expecting a set_i action to occur, so they could proceed by setting $x = i$ and then perform $check_i$ action and end up in the critical section. This might lead to a bad interleaving if processes' set_i and $check_i$ actions take place sequentially so that each one ends up in the critical section. In order to avoid this bad interleaving we need to ensure that all the processes that performed $test_i$ action successfully will perform set_i before any process can perform $check_i$. This is done through setting $last\_main(i) := now + u\_main$ when a process successfully completes the $text_i$ action. This then sets the deadline for the set_i action to complete. It is ensured through the stopping condition of the trajectory. When a set_i action takes place, along with changing the turn variable value we also set $first\_check(i) := now + l\_check$. This is a time constraint for the earliest time a $check_i$ action becomes enabled, hence, transition can take place. Once we ensure that $u\_main < l\_check$ we guarantee that there will be no bad interleaving, but rather only the process last to execute its set_i action will proceed to the critical section.

The fischer_me.tioa captures the above algorithm in the TIOA format and is pretty straightforward. The only ambiguity was using $AugmentedReal$ datatype rather than $Real$ for the $last\_main(i)$ since it was supposed to be set to infinity after the set_i action. We also had to explicitly state non negativity, ordering of the bounds and that $u\_main < l\_check$.

Having developed fischer_me.tioa we used the `tioa2pvs` translator which is a part of the TIOA toolkit. Translation completed in a matter of seconds and produced a set of PVS theories specifying our automaton and its properties: fishcer_me_decls.pvs, common_decls.pvs, fischer_me_invariants. The first file contained the description of the automaton with the transitions, actions and constraints. The second file imports the definitions necessary for the trajectory notion of time and variable *now* from the TAME libraries. The third file contains a list of invariants that we had to prove translated from their TIOA equivalents.

The main invariant that states the mutex property is the following: *There doesn't exist i and j, $i \neq j$, such that $pc(i) = pc(j) = pc\_crit$.* However, in order to guarantee mutex we needed an auxiliary property: *If a process i is in critical section then no other process will be able to set the turn variable x, hence, it will retain value i until the process exits critical section.* These two properties guarantee mutual exclusion. However, in order to prove them successfully we needed to add several other invariants as follows:

1. *now* and $first\_check(i)$ are non negative
2. If an action other than *pc_check* took place then $last\_main(i)$ has the value less or equal to $now + u\_main$ (upper bound of main tasks). This imposes the time bound condition.
3. *now* is always less than $last\_main(i)$
4. If some process i is not in *pc_check* then its $last\_main(i) \neq \infty$
5. If processes pass test simultaneously then all set_i will take place before the first $check_i$

These invariants imply bounds and ordering of different events and induced from the specification. Each invariant was proved in just several straightforward steps. After these invariants were proved they could be used explicitly by the verifier during the proof of the two mutex invariants that we were primarily interested in. Since we proved all the auxiliary invariants that we could have needed it was just a matter of using *induction* strategy, referring to proved properties, and *grinding* through the proof. The details of the proof and the proof tree can be found in mutex.prf.

A.1 fischer_me.tioa

```
% Fischer's mutual exclusion algorithm
% Composite version with upper and lower bounds
% Taken from example 4.5 (figure 5 and 6) in [KLSV2004],
% includes task bounds described in section 24.2
% of Distributed Algorithms [Lynch].
%
% This file contains invariants we want to prove as
% well as the description of the automaton.
%
% All processes have a positive index, as enforced in
% the invariants as "∀ (k: Int) ((k > 0) ⇒ Invariant)"
% and in the preconditions of actions.
% "0" is used as a marker for the variable turn when it
% is not set.

%% type declaration %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

type PcValue = enumeration of
  pc_rem,
  pc_test,
  pc_set,
  pc_check,
  pc_leavetry,
```

```
  pc_crit,
  pc_leaveexit,
  pc_reset


%% description of the automata %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

automaton fischer_me(l_main, l_check: Real, u_main, u_check: Real)

  signature
    output
      try(i: Int),
      crit(i: Int),
      exit(i: Int),
      rem(i: Int)
    internal
      test(i: Int),
      set(i: Int),
      check(i: Int),
      reset(i: Int)

  states
    turn: Int := 0,
    now: Real := 0,
    pc: Array[Int, PcValue],
    first_main: Array[Int, Real],
    last_main: Array[Int, AugmentedReal],
    first_check: Array[Int, Real],
    last_check: Array[Int, AugmentedReal]

    initially

                    % non negativity
                    u_main ≥ int2real(0) ∧
                    u_check ≥ int2real(0) ∧
                    l_main ≥ int2real(0) ∧
                    l_check > int2real(0) ∧

                    % order
                    l_main ≤ u_main ∧ l_check ≤ u_check ∧

                    % condition that set and other actions has to occur before check
                    u_main < l_check ∧

    % start states
    ∀ i: Int (
                    pc[i] = pc_rem ∧
                    first_main[i] = l_main ∧
                    last_main[i] = u_main ∧
                    first_check[i] = int2real(0) ∧
                    last_check[i] = ∈fty


                    )
```

```
transitions

  output try(i)
    pre
      pc[i] = pc_rem  ∧ i > 0
    eff
      pc[i] := pc_test;

  internal test(i)
    pre
      pc[i] = pc_test ∧ first_main[i] ≤ now ∧ i > 0
    eff
      if turn = 0 then
        pc[i] := pc_set;
        first_main[i] := now + l_main;
        if ((now + u_main) ≥ int2real(0)) then
          last_main[i] := real2augmented(now + u_main);
        fi
      fi

  internal set(i)
    pre
      pc[i] = pc_set ∧ first_main[i] ≤ now ∧ i > 0
    eff
      turn := i;
      pc[i] := pc_check;
      first_main[i] := 0;
      last_main[i] := ∈fty;
      first_check[i] := now + l_check;
      if ((now + u_check) ≥ int2real(0)) then
        last_check[i] := real2augmented(now + u_check);
      fi

  internal check(i)
    pre
      pc[i] = pc_check ∧ first_check[i] ≤ now ∧ i > 0
    eff
      if turn = i then
        pc[i] := pc_leavetry;
      else
        pc[i] := pc_test;
      fi;
      first_main[i] := now + l_main;
      first_check[i] := 0;
      last_check[i] := ∈fty;
      if ((now + u_main) ≥ int2real(0)) then
        last_main[i] := real2augmented(now + u_main);
      fi

  output crit(i)
    pre
      pc[i] = pc_leavetry ∧ first_main[i] ≤ now ∧ i > 0
    eff
      pc[i] := pc_crit;
      first_main[i] := now + l_main;
```

```
        if ((now + u_main) ≥ int2real(0)) then
          last_main[i] := real2augmented(now + u_main);
        fi

    output exit(i)
      pre
        pc[i] = pc_crit ∧ i > 0
      eff
        pc[i] := pc_reset;

    internal reset(i)
      pre
        pc[i] = pc_reset ∧ first_main[i] ≤ now ∧ i > 0
      eff
        first_main[i] := now + l_main;
        pc[i] := pc_leaveexit;
        turn := 0;
        if ((now + u_main) ≥ int2real(0)) then
          last_main[i] := real2augmented(now + u_main);
        fi

    output rem(i)
      pre
        pc[i] = pc_leaveexit ∧ first_main[i] ≤ now ∧ i > 0
      eff
        first_main[i] := now + l_main;
        pc[i] := pc_rem;
        if ((now + u_main) ≥ int2real(0)) then
          last_main[i] := real2augmented(now + u_main);
        fi

  trajectories
    trajdef traj
      stop when
        ∃ i:Int
          (now ≥ int2real(0) ∧
           (now = last_main[i] ∨
            now = last_check[i]))
      evolve
        d(now)= 1

%%%%%%%%% invariants we want to prove %%%%%%%%%%%%%%%%%%%%%%%%%%%%

invariant of fischer_me:
  % non negativity
  u_main ≥ int2real(0) ∧ u_check ≥ int2real(0) ∧ l_main ≥ int2real(0)
  ∧ l_check > int2real(0)
  ∧
  % order
  l_main ≤ u_main ∧ l_check ≤ u_check ∧
  % main and check condition
  u_main < l_check


invariant of fischer_me:
```

```
  ∀ k:Int (now≥int2real(0) ∧ first_check[k] ≥ int2real(0))

invariant of fischer_me:
  ∀ k:Int ((k>0 ∧ (now + u_main) ≥ int2real(0) ∧
            pc[k]≠pc_check)
           ⇒ (last_main[k]≤(now+u_main)))

invariant of fischer_me:
  ∀ k:Int ((k>0 ∧ now ≥ int2real(0) )
           ⇒ (now≤last_main[k]))

invariant of fischer_me:
  ∀ k:Int ((k>0 ∧ now ≥ int2real(0) ∧ pc[k] ≠ pc_check)
           ⇒ (last_main[k] ≠ ∈fty))

invariant of fischer_me:
  ∀ i: Int ∀ j: Int (
                  (i > 0 ∧ j > 0 ∧ first_check[i] ≥ int2real(0) ∧
                   pc[i] = pc_check ∧
                   turn = i ∧
                   pc[j] = pc_set)
                  ⇒ (first_check[i]>last_main[j]))

invariant of fischer_me:
  ∀ i:Int ∀ j: Int (
           ((i > 0 ∧ j > 0) ∧
            (pc[i] = pc_leavetry ∨
             pc[i] = pc_crit ∨
             pc[i] = pc_reset))
           ⇒ (turn=i ∧ pc[j]≠pc_set))

invariant of fischer_me:
     ∀ i: Int ∀ j: Int (
                    (i > 0 ∧ j > 0 ∧ i ≠ j)
                    ⇒ ((pc[i] ≠ pc_crit) ∨  (pc[j] ≠ pc_crit)))
```

A.2 fischer_me_decls.pvs

```
%% fischer_me_decls.pvs %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% This file is generated by the TIOAtoPVS translator.
%% It contains a description of the automaton for Fischer Mutex.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fischer_me_decls : THEORY
BEGIN

IMPORTING common_decls


% Automaton formal parameters
  l_main: real
  l_check: real
  u_main: real
  u_check: real
```

```
%  Tuples, Enums and Unions
PcValue : TYPE = {pc_rem, pc_test, pc_set, pc_check, pc_leavetry, pc_crit,
                  pc_leaveexit, pc_reset}


% States variables declarations
states: TYPE = [#
  turn: int,
  now: real,
  pc: array[int -> PcValue],
  first_main: array[int -> real],
  last_main: array[int -> time],
  first_check: array[int -> real],
  last_check: array[int -> time]
#]


% Start state definition
start (s:states):bool = (LAMBDA(s: states):
  turn(s) = 0 AND
  now(s) = 0 AND
  (((((((u_main >= 0) AND (u_check >= 0)) AND (l_main >= 0)) AND (l_check > 0))
    AND (l_main <= u_main)) AND (l_check <= u_check)) AND (u_main < l_check))
    AND FORALL(i: int) :  ((((((pc(s)(i) = pc_rem) AND (first_main(s)(i) = l_main))
    AND (last_main(s)(i) = fintime(u_main))) AND (first_check(s)(i) = 0))
    AND (last_check(s)(i) = infinity))))
)(s);

% Action declarations
interval(i, j: (fintime?)): TYPE = {s: (fintime?) | i <= s AND s <= j AND i <= j}
f_type(i, j: (fintime?)): TYPE = [interval(i, j)->states]

% actions signatures
actions: DATATYPE
BEGIN
  nu_traj(delta_t: {t: (fintime?) | dur(t)>=0}, F:f_type(zero, delta_t)): nu_traj?
  try(i: int) : try?
  crit(i: int) : crit?
  exit(i: int) : exit?
  rem(i: int) : rem?
  test(i: int) : test?
  set(i: int) : set?
  check(i: int) : check?
  reset(i: int) : reset?
END actions


% actions visibility
visible(a:actions): bool =
CASES a OF
  nu_traj(delta_t, F): true,
  try(i) : true,
  crit(i) : true,
  exit(i) : true,
  rem(i) : true,
  test(i) : false,
  set(i) : false,
  check(i) : false,
  reset(i) : false
```

```
ENDCASES

% time passage actions
timepassageactions(a:actions): bool =
CASES a OF
  nu_traj(delta_t, F): true,
  try(i) : false,
  crit(i) : false,
  exit(i) : false,
  rem(i) : false,
  test(i) : false,
  set(i) : false,
  check(i) : false,
  reset(i) : false
ENDCASES


% Transition definitions
% preconditions
enabled_specific (a:actions, s:states):bool = CASES a OF

  nu_traj(delta_t, F):  FORALL (t:interval(zero,delta_t)):

    % Stop condition
    ((EXISTS(i: int) :  (((now(F(t)) >= 0)
         AND ((fintime(now(F(t))) = last_main(F(t))(i))
               OR (fintime(now(F(t))) = last_check(F(t))(i)))))
                 ) IMPLIES t = delta_t) AND

    % Evolve predicate
    F(t) = s WITH [
      now := now(s) + 1 * dur(t)],

  try(i):
    ((pc(s)(i) = pc_rem) AND (i > 0)),

  crit(i):
    (((pc(s)(i) = pc_leavetry) AND (first_main(s)(i) <= now(s))) AND (i > 0)),

  exit(i):
    ((pc(s)(i) = pc_crit) AND (i > 0)),

  rem(i):
    (((pc(s)(i) = pc_leaveexit) AND (first_main(s)(i) <= now(s))) AND (i > 0)),

  test(i):
    (((pc(s)(i) = pc_test) AND (first_main(s)(i) <= now(s))) AND (i > 0)),

  set(i):
    (((pc(s)(i) = pc_set) AND (first_main(s)(i) <= now(s))) AND (i > 0)),

  check(i):
    (((pc(s)(i) = pc_check) AND (first_check(s)(i) <= now(s))) AND (i > 0)),

  reset(i):
    (((pc(s)(i) = pc_reset) AND (first_main(s)(i) <= now(s))) AND (i > 0))


ENDCASES
```

```
enabled (a:actions, s:states):bool = enabled_specific(a,s)

% effects
% transitions effects using substitution
trans (a:actions, s:states):states = CASES a OF

  nu_traj(delta_t, F): F(delta_t),

  try(i):
    s WITH [
      pc :=
        (pc(s) WITH [(i) := pc_test])
    ],

  crit(i):
    s WITH [
      first_main :=
        (first_main(s) WITH [(i) := (now(s) + l_main)]),

      pc :=
        (pc(s) WITH [(i) := pc_crit]),

      last_main :=
        (IF ((now(s) + u_main) >= 0) THEN
            (last_main(s) WITH [(i) := fintime(now(s) + u_main)])
          ELSE
            last_main(s)
          ENDIF )
    ],

  exit(i):
    s WITH [
      pc :=
        (pc(s) WITH [(i) := pc_reset])
    ],

  rem(i):
    s WITH [
      first_main :=
        (first_main(s) WITH [(i) := (now(s) + l_main)]),

      pc :=
        (pc(s) WITH [(i) := pc_rem]),

      last_main :=
        (IF ((now(s) + u_main) >= 0) THEN
            (last_main(s) WITH [(i) := fintime(now(s) + u_main)])
          ELSE
            last_main(s)
          ENDIF )
    ],

  test(i):
    s WITH [
      first_main :=
        (IF (turn(s) = 0) THEN
            (first_main(s) WITH [(i) := (now(s) + l_main)])
          ELSE
            first_main(s)
```

```
        ENDIF ),

    pc :=
      (IF (turn(s) = 0) THEN
          (pc(s) WITH [(i) := pc_set])
        ELSE
          pc(s)
        ENDIF ),

    last_main :=
      (IF (turn(s) = 0) THEN
          (IF ((now(s) + u_main) >= 0) THEN
              (last_main(s) WITH [(i) := fintime(now(s) + u_main)])
            ELSE
              last_main(s)
            ENDIF )
        ELSE
          last_main(s)
        ENDIF )
  ],

set(i):
  s WITH [
    last_check :=
      (IF ((now(s) + u_check) >= 0) THEN
          (last_check(s) WITH [(i) := fintime(now(s) + u_check)])
        ELSE
          last_check(s)
        ENDIF ),

    first_check :=
      (first_check(s) WITH [(i) := (now(s) + l_check)]),

    first_main :=
      (first_main(s) WITH [(i) := 0]),

    pc :=
      (pc(s) WITH [(i) := pc_check]),

    turn := i,

    last_main :=
      (last_main(s) WITH [(i) := infinity])
  ],

check(i):
  s WITH [
    last_check :=
      (last_check(s) WITH [(i) := infinity]),

    first_check :=
      (first_check(s) WITH [(i) := 0]),

    first_main :=
      (first_main(s) WITH [(i) := (now(s) + l_main)]),

    pc :=
      (IF (turn(s) = i) THEN
          (pc(s) WITH [(i) := pc_leavetry])
        ELSE
```

```
            (pc(s) WITH [(i) := pc_test])
          ENDIF ),

      last_main :=
        (IF ((now(s) + u_main) >= 0) THEN
            (last_main(s) WITH [(i) := fintime(now(s) + u_main)])
          ELSE
            last_main(s)
          ENDIF )
    ],

  reset(i):
    s WITH [
      first_main :=
        (first_main(s) WITH [(i) := (now(s) + l_main)]),

      pc :=
        (pc(s) WITH [(i) := pc_leaveexit]),

      turn := 0,

      last_main :=
        (IF ((now(s) + u_main) >= 0) THEN
            (last_main(s) WITH [(i) := fintime(now(s) + u_main)])
          ELSE
            last_main(s)
          ENDIF )
    ]


ENDCASES


% effects
% transitions effects using LET
trans2 (a:actions, s:states):states = CASES a OF

  nu_traj(delta_t, F): F(delta_t),


  try(i):
    (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_test]] IN s),

  crit(i):
    (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_crit]] IN
    (LET s:states = s WITH [first_main := first_main(s)
                            WITH [(i) := (now(s) + l_main)]] IN
    (LET s:states =
      IF ((now(s) + u_main) >= 0) THEN
        (LET s:states = s WITH [last_main := last_main(s)
                                WITH [(i) := fintime(now(s) + u_main)]] IN s)
      ELSE s
      ENDIF
    IN s))),

  exit(i):
    (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_reset]] IN s),

  rem(i):
    (LET s:states = s WITH [first_main := first_main(s) WITH [(i) := (now(s) + l_main)]] IN
```

```
  (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_rem]] IN
  (LET s:states =
    IF ((now(s) + u_main) >= 0) THEN
      (LET s:states = s WITH [last_main := last_main(s)
                         WITH [(i) := fintime(now(s) + u_main)]] IN s)
    ELSE s
    ENDIF
  IN s))),

test(i):
  (LET s:states =
    IF (turn(s) = 0) THEN
      (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_set]] IN
      (LET s:states = s WITH [first_main := first_main(s)
                         WITH [(i) := (now(s) + l_main)]] IN
      (LET s:states =
        IF ((now(s) + u_main) >= 0) THEN
          (LET s:states = s WITH [last_main := last_main(s)
                             WITH [(i) := fintime(now(s) + u_main)]] IN s)
        ELSE s
        ENDIF
      IN s)))
    ELSE s
    ENDIF
  IN s),

set(i):
  (LET s:states = s WITH [turn := i] IN
  (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_check]] IN
  (LET s:states = s WITH [first_main := first_main(s) WITH [(i) := 0]] IN
  (LET s:states = s WITH [last_main := last_main(s) WITH [(i) := infinity]] IN
  (LET s:states = s WITH [first_check := first_check(s) WITH [(i) := (now(s) + l_check)]] IN
  (LET s:states =
    IF ((now(s) + u_check) >= 0) THEN
      (LET s:states = s WITH [last_check := last_check(s)
                         WITH [(i) := fintime(now(s) + u_check)]] IN s)
    ELSE s
    ENDIF
  IN s))))))),

check(i):
  (LET s:states =
    IF (turn(s) = i) THEN
      (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_leavetry]] IN s)
    ELSE
      (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_test]] IN s)
    ENDIF
  IN
  (LET s:states = s WITH [first_main := first_main(s)
                     WITH [(i) := (now(s) + l_main)]] IN
  (LET s:states = s WITH [first_check := first_check(s) WITH [(i) := 0]] IN
  (LET s:states = s WITH [last_check := last_check(s) WITH [(i) := infinity]] IN
  (LET s:states =
    IF ((now(s) + u_main) >= 0) THEN
      (LET s:states = s WITH [last_main := last_main(s)
                         WITH [(i) := fintime(now(s) + u_main)]] IN s)
    ELSE s
    ENDIF
  IN s))))),
```

```
  reset(i):
    (LET s:states = s WITH [first_main := first_main(s)
                        WITH [(i) := (now(s) + l_main)]] IN
    (LET s:states = s WITH [pc := pc(s) WITH [(i) := pc_leaveexit]] IN
    (LET s:states = s WITH [turn := 0] IN
    (LET s:states =
      IF ((now(s) + u_main) >= 0) THEN
        (LET s:states = s WITH [last_main := last_main(s)
                           WITH [(i) := fintime(now(s) + u_main)]] IN s)
      ELSE s
      ENDIF
    IN s))))

ENDCASES

% Import statements
IMPORTING timed_auto_lib@time_machine[states, actions, enabled, trans, start,
   visible, timepassageactions,
   (lambda(a:({x:actions|timepassageactions(x)})): dur(delta_t(a)))]

END fischer_me_decls
```

A.3 fischer_me_invariants.pvs

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Invariants for automaton fischer_me
%% This file is automatically generated by the TIOAtoPVS translator.
%% It contains invariants to prove the mutex property.
%% lemma 7 states the mutex property.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fischer_me_invariants: THEORY

BEGIN

  IMPORTING fischer_me_decls
  IMPORTING fischer_me_rewrite_aux_1
  IMPORTING fischer_me_rewrite_aux_2
  IMPORTING fischer_me_unique_aux

  Inv_0(s:states):bool =
    (((((((u_main >= 0) AND (u_check >= 0)) AND (l_main >= 0)) AND (l_check > 0))
        AND (l_main <= u_main)) AND (l_check <= u_check)) AND (u_main < l_check))

  lemma_0: LEMMA (FORALL (s:states): reachable(s) => Inv_0(s));

  Inv_1(s:states):bool =
    FORALL(k: int) :  (((now(s) >= 0) AND (first_check(s)(k) >= 0)))

  lemma_1: LEMMA (FORALL (s:states): reachable(s) => Inv_1(s));

  Inv_2(s:states):bool =
    FORALL(k: int) :  (((((k > 0) AND ((now(s) + u_main) >= 0)) AND (pc(s)(k) /= pc_check))
         => (last_main(s)(k) <= fintime(now(s) + u_main))))

  lemma_2: LEMMA (FORALL (s:states): reachable(s) => Inv_2(s));

  Inv_3(s:states):bool =
    FORALL(k: int) :  ((((k > 0) AND (now(s) >= 0)) => (fintime(now(s)) <= last_main(s)(k))))
```

```
  lemma_3: LEMMA (FORALL (s:states): reachable(s) => Inv_3(s));

  Inv_4(s:states):bool =
    FORALL(k: int) :  (((((k > 0) AND (now(s) >= 0)) AND (pc(s)(k) /= pc_check))
            => (last_main(s)(k) /= infinity)))

  lemma_4: LEMMA (FORALL (s:states): reachable(s) => Inv_4(s));

  Inv_5(s:states):bool =
    FORALL(i: int) :  (FORALL(j: int) :  (((((((i > 0) AND (j > 0))
          AND (first_check(s)(i) >= 0)) AND (pc(s)(i) = pc_check))
          AND (turn(s) = i)) AND (pc(s)(j) = pc_set))
               => (fintime(first_check(s)(i)) > last_main(s)(j)))))

  lemma_5: LEMMA (FORALL (s:states): reachable(s) => Inv_5(s));

  Inv_6(s:states):bool =
    FORALL(i: int) :  (FORALL(j: int) :  (((((i > 0) AND (j > 0))
           AND (((pc(s)(i) = pc_leavetry) OR (pc(s)(i) = pc_crit))
              OR (pc(s)(i) = pc_reset)))
                  => ((turn(s) = i) AND (pc(s)(j) /= pc_set)))))

  lemma_6: LEMMA (FORALL (s:states): reachable(s) => Inv_6(s));

  Inv_7(s:states):bool =
    FORALL(i: int) :  (FORALL(j: int) :  (((((i > 0) AND (j > 0)) AND (i /= j))
            => ((pc(s)(i) /= pc_crit) OR (pc(s)(j) /= pc_crit)))))

  lemma_7: LEMMA (FORALL (s:states): reachable(s) => Inv_7(s));

END fischer_me_invariants
```

A.4 common_decls.pvs

```
%% This file just imports the definition needed for fischer_me_decls.pvs
%% fischer_me_decls.pvs imports this file.

common_decls:THEORY
BEGIN

   timed_auto_lib: LIBRARY = "../timed_auto_lib"

   IMPORTING timed_auto_lib@time_thy
   IMPORTING timed_auto_lib@list_rewrites
   IMPORTING timed_auto_lib@bool_rewrites

END common_decls
```

B TwoTaskRace example

In the two-task race system, the automaton `TwoTaskRace` increments `count` until it is interrupted by a `set` action. Thereafter, it decrements `count` and reports when `count` reaches 0. `first_main` and `last_main` records the earliest and latest times when an action in the set {`increment`, `decrement`, `report`} can occur. `first_set` and `last_set` records the earliest and latest times the action `set` can occur. The preconditions enforce the start times of each action, while the trajectory definitions enforce the deadlines.

To prove that the `report` action occurs within a certain time bound, we create an abstract automaton `TwoTaskRaceSpec` which simply performs a `report` action within the time bounds `first_report` and

```
1 TwoTaskRace.reported = TwoTaskRaceSpec.reported ∧
2 TwoTaskRace.now = TwoTaskRaceSpec.now ∧
3 (¬TwoTaskRace.flag ∧ TwoTaskRace.last_main < TwoTaskRace.first_set ⇒
4   TwoTaskRaceSpec.first_report ≤
5     (min(TwoTaskRace.first_set, TwoTaskRace.first_main) +
6       ((int2real(TwoTaskRace.count) +
7         ((TwoTaskRace.first_set - TwoTaskRace.last_main) / a2)) * a1)))
8 ∧
9 (TwoTaskRace.flag ∨ TwoTaskRace.last_main ≥ TwoTaskRace.first_set ⇒
10  TwoTaskRaceSpec.first_report ≤
11    (TwoTaskRace.first_main + (int2real(TwoTaskRace.count) * a1)))
12 ∧
13 (¬TwoTaskRace.flag ∧ TwoTaskRace.first_main ≤ TwoTaskRace.last_set ⇒
14  TwoTaskRaceSpec.last_report ≥
15    (TwoTaskRace.last_set +
16      ((int2real(TwoTaskRace.count + 2) +
17        ((TwoTaskRace.last_set - TwoTaskRace.first_main) / a1)) * a2)))
18 ∧
19 (¬(TwoTaskRace.reported) ∧
20  (TwoTaskRace.flag ∨ TwoTaskRace.first_main > TwoTaskRace.last_set) ⇒
21  TwoTaskRaceSpec.last_report ≥
22    (TwoTaskRace.last_main + (int2real(TwoTaskRace.count) * a2)))
```

```
1 ((reported(s_A) = reported(s_B)) ∧
2 (now(s_A) = now(s_B))) ∧
3 (((¬ flag(s_A) ∧ (last_main(s_A) < first_set(s_A))) ⇒
4     (first_report(s_B) ≤
5       min(first_set(s_A), first_main(s_A)) +
6       (count(s_A) + (first_set(s_A) − last_main(s_A))/a2) × a1)))
7 ∧
8 (((flag(s_A) ∨ (last_main(s_A) ≥ first_set(s_A))) ⇒
9     (first_report(s_B) ≤ first_main(s_A) + count(s_A) × a1)))
10 ∧
11 (((¬ flag(s_A) ∧ (first_main(s_A) ≤ last_set(s_A))) ⇒
12     last_report(s_B) ≥
13       last_set(s_A) +
14       (count(s_A) + 2 + (last_set(s_A) − first_main(s_A))/a1) × a2))
15 ∧
16 (((¬ reported(s_A) ∧
17       (flag(s_A) ∨ (first_main(s_A) > last_set(s_A)))) ⇒
18     last_report(s_B) ≥ last_main(s_A) + count(s_A) × a2))
```

Figure 11: TIOA and PVS descriptions of the simulation relation

`last_report`. We then prove a forward simulation relation from the concrete automaton `TwoTaskRace` to the abstract automaton `TwoTaskRaceSpec`. The values of the two time bounds are as follows:

```
first_report: Real := IF (a2<b1) THEN min(b1,a1) + (((b1-a2)*a1)/a2) ELSE a1,
last_report: AugmentedReal := real2augmented(b2 + a2 + ((b2 * a2) / a1))
```

In addition to the concrete automaton `TwoTaskRace`, we write the abstract automaton `TwoTaskRaceSpec`, the invariants of both automata, and the forward simulation relation in TIOA, and obtain the PVS translation through the translator. Using PVS, we prove the invariants of each automaton, and then prove the simulation relation. Figure 11 shows the six conjunctions of the simulation relation. In the TIOA code, lines 1–2 asserts that the variables `reported` and `now` are equal in both automata. Lines 3–11 relates the lower bound `first_report` while lines 13–22 relates the upper bound `last_report`. Figure 11 also shows the PVS code. s_A and s_B refer to the state of the concrete and abstract automata respectively.

The proof of the simulation relation involves using induction and performing case splits on the actions and verifying the inequalities in the relation. The induction hypothesis assumes that a pre-state $\mathbf{x}_{\mathcal{A}}$ of the concrete automaton $\mathcal{A}$ is related to a pre-state $\mathbf{x}_{\mathcal{B}}$ of the abstract automaton $\mathcal{B}$. If the action $a_{\mathcal{A}}$ is an external action or a time passage action, we show the existence of a corresponding action $a_{\mathcal{B}}$ in $\mathcal{B}$ such that the $a_{\mathcal{B}}$ is enabled in $\mathbf{x}_{\mathcal{B}}$ and that the post-states obtained by performing $a_{\mathcal{A}}$ and $a_{\mathcal{B}}$ are related. If the action $a_{\mathcal{A}}$ is internal, we show that the post-state of $a_{\mathcal{A}}$ is related to $\mathbf{x}_{\mathcal{B}}$. To show that two states are related, we prove the six conjunctions in the relation using invariants of each automaton.

C TIOA Simulator Syntax

C.1 TIOA-sim

The syntax of TIOA-sim is the same as that of TIOA. The restrictions we impose are indicated below:

- Let `{v1, ... , vn}` be the set state variables of type `Real` that are not discrete. In each trajectory definition, the **evolve** clause consits of a list of n equations
  `d(v1) = exp1 ; ... d(vn)= expn`

such that `{exp1, ..., expn}` are constant valued expressions.

- If a stopping condition for a trajectory definition exists then it must be a disjunction of equalities of the form `v=exp` where `v` is a variable of type `Real` that is not discrete and `exp` is an expression of type `discrete Real` or `discrete AugmentedReal`. The type constraint ensures that the value of each `exp` remains constant throughout a trajectory.

C.2 Syntax for resolving Nondeterminism

Extensions to Primitive Automaton Syntax

```
simpleBody  :=  'signature' formalActionList+ states
                transitions trajectories?  tasks?  schedule?
schedule    :=  'schedule' states?  'do' NDRProgram 'od'
```

Extensions to Choice Syntax

```
choice     :=  'choose' (variable ('where' predicate)?)?  choiceNDR?
choiceNDR  :=  'det' 'do' NDRProgram 'od' | NDRYield
```

NDR Programs

```
NDRProgram       ::=  NDRStatement;*
NDRStatement     ::=  assignment | NDRConditional | NDRWhile
                      | NDRYield | NDRFire | NDRFollowTraj
NDRConditional   ::=  'if' predicate 'then' NDRProgram
                      ('elseif' predicate 'then' NDRProgram)*
                      ('else' NDRProgram)?  'fi'
NDRWhile         ::=  'while' predicate 'do' NDRProgram 'od'
NDRFire          ::=  'fire' actionType actionName actionActuals?  transCase?
                      | 'fire'
NDRYield         ::=  'yield' term
NDRFollowTraj    ::=  'follow trajectory' trajdefName 'duration' term
```

C.3 Syntax for Paired Simulation

```
simulation           ::=  ('forward' | 'backward')
simProof             ::=  'proof' states?  ('start' (lvalue ':=' term);+)?
                          simProofEntry+
simProofEntry        ::=  simProofTransEntry | simProofTrajEntry
simProofTransEntry   ::=  'for' actionType actionName actionFormals?  transCase?
                          (('do' proofProgram 'od') | 'ignore')
simProofTrajEntry    ::=  'for' 'trajectory' trajdefName 'duration' durationVar
                          (('do' proofProgram 'od') | 'ignore')
proofProgram         ::=  simProofStatement;+
simProofStatement    ::=  assignment
                          | simProofConditional
                          | simProofWhile
                          | simProofFire
                          | simProofFollow
simProofFollow       ::=  'follow' trajName '(' term ')'
```

D Phase II overview

In this appendix we overview the work envisioned for Phase II of the project.

D.1 Summary of the proposed Phase II project

The results of Phase I show the feasibility of developing a robust, commercial-grade version of the toolset suitable for modeling and analyzing real distributed systems. In Phase II, we will develop a comprehensive and integrated design environment supporting the TIOA formalism. Here we briefly identify the Phase II activites. We give the objectives of the project in more detail in Section D.2.

Three areas of development together constitute the proposed Phase II project:

1. TIOA language and its formal framework: The language specification and the framework for reasoning about TIOA specifications have been largely developed in Phase I. In Phase II we will finalize this development and produce comprehensive user documentation.
2. Computer-aided design tools supporting the development and reasoning about specifications of complex distributed systems expressed in the TIOA formalism. These robust, end-user oriented computer-aided design tools will be based on the proof-of-concept implementations developed in Phase I and interface with other existing tools:
   - Front end language processor, `tioaCheck`, designed to accept the TIOA specification, perform static and type analysis of the specification, and produce intermediate output for use by other tools.
   - The simulator, `tioaSim`, designed to simulate executions of TIOA specifications and to provide linked simulations of pairs of specifications, where one specification gives an abstract definition and the other is a more concrete specification that is supposed to implement the abstract definition.
   - A model checking component will be developed by the academic partners Grosu and Smolka at Stony Brook University. This will extend their independently developed technique of Monte Carlo model checking to the TIOA implementation problem. The work will involve the translation of TIOA specifications to the input language of the Open-Kronos model checker for timed automata. Monte Carlo model checking has already been implemented in Open-Kronos and has demonstrated significant performance and scalability advantages.
   - Interface to PVS theorem proving. We will implement the translator, called `tioa2pvs`, based on the prototype developed in Phase I. The translator accepts output from `tioaCheck` and produces native PVS specifications. We will also provide a suite of PVS strategies carefully designed for use with translated TIOA specifications.
3. Integration, GUI, and usability features:
   - We will integrate the computer-aided tools to provide a development environment for end-users.
   - The environment will include an integrated GUI based on the Eclipse framework (www.eclipse.org). We will also provide facilities for visual representation of specifications and for graphical representation of the results, for example, for visualizing simulated executions within `tioaSim`.
   - We will development a complete set of user documents consisting of reference manuals, user guides, and documentation of the overall toolset usage patterns.

   The project will also include quality-assurance activities that will not have specific corresponding deliverables, but that will be designed to improve the quality and usability of the final product.

D.2 Phase II Technical Objectives

Here we enumerate the specific Phase II objectives and other existing tools and prior development that form the basis for Phase II deliverables.

1. **The TIOA (timed input/output automata) Language**

   The TIOA model provides a precise and expressive linguistic framework for developing specifications for a broad class of systems, including systems that involve distributed, communicating, concurrent, and timed components. In Phase II are going to formalize the language definition and the mathematical framework supporting formal reasoning about specifications and algorithms expressed in TIOA.

   In Phase I we essentially completed the development of the TIOA language and its mathematical framework. The refinements planned for Phase II will primarily deal with syntactical and packaging issues driven by the needs of the tools being developed within the project, and improving the language presentation for its users.

2. **Language Front End**

   The front end makes the mathematical TIOA language suitable for use with computer-aided design tools. This includes the ability to parse a program specified in TIOA and to represent the semantics of the program relevant to the computer-aided tools.

   In Phase I we developed a fully functional prototype of the front end, called `tioaCheck`. Phase II work will focus on transforming the front end into a robust tool, with substantially improved usability features, such as integrated GUI and editing facility, and comprehensive diagnostics. We are also going to refine the output formats produced by the front end to implement better traceability and ease of integration with other tools.

3. **Simulator**

   A major deliverable of this project is the TIOA language simulator, `tioaSim`. The simulator serves two purposes: (1) It allows the users to "run" sample executions of a TIOA program on a single machine; (2) It allows the users to run *paired* simulations of two TIOA programs, where a function is provided to map the states of one program to another.

   In Phase I we developed a prototype simulator, which was substantially tested for single automaton simulation. We have also implemented the paired simulation functionality. Currently we are testing it with sample automata. Phase II work will focus on (a) making the simulator into a robust end-user system, (b) adding traceability features, (c) integrating it with the overal GUI framework, and (d) implementing a visualization strategy allowing the users to develop their own visualizations.

4. **Model Checking**

   Model checking [14, 90], the problem of deciding whether or not a property specified in temporal logic holds of a system specification, has gained wide acceptance within the hardware and protocol verification communities, and is witnessing increasing application in the domain of software verification. The beauty of this technique is that when the state space of the system under investigation is finite-state, model checking may proceed in a fully automatic, push-button fashion. Moreover, should the system fail to satisfy the formula, a counter-example trace leading the user to the error state is produced.

   Model checking is related to the *implementation problem* for TIOA, which is the following. Given timed I/O automata A and B, representing the implementation and specification of the system under investigation, does A implement B ($A \leq B$)? If a TIOA's signature and state space (except for the time dimension) are finite, then it can be regarded as a (input-enabled) *timed automaton* [1]. Therefore, model-checking techniques for timed automata (TA) can be brought to bear on the TIOA implementation problem. For the Phase II project, we intend to build on our prior work on Monte Carlo model checking [34] by extending

this approach to TIOA. This work was not part of the Phase I effort, but rather was developed independently by Stony Brook University subcontractors Grosu and Smolka. Our initial results on applying Monte Carlo model checking to TA are very encouraging; this leads us to believe that we will witness the same performance and scalability advantages offered by this approach during the Phase II effort.

5. **PVS and theorem proving**

   In Phase I we have established that TIOA language is well-suited for translation into axioms that can be used by existing interactive theorem provers, and specifically PVS. In such translations, all imperative statements in the effects of transition definitions, including assignment statements, choose statements, conditionals, and loops, are replaced by functions and predicates relating poststates to prestates. Other axioms are derived from formal definitions of the data types used in the automata. We have used theorem provers to prove properties of data types used in automata, invariants of automata, and simulation relations between automata. We are able to identify and formulate proof obligations for cases in which validity requirements for TIOA programs cannot be checked statically by the front end.

   In Phase II we are going to develop a production-grade interface to PVS and provide PVS strategies specifically designed to deal with specifications derived from TIOA. The motivation for developing a TIOA-specific interface for PVS is two fold: first, it will relieve the TIOA users from having to master PVS's own specification language and its quirks; second, with a suite of TIOA specific PVS proof strategies, it will be possible to limit the human interaction to the essential parts, and also to generate human readable proofs.

6. **Component integration**

   The overall goal of work on sofware tools in Phase II is to raise the current prototype tools to production quality. We expect to gain the following benefits including improved user interfaces for the TIOA tools, improved integration of the TIOA tools, extended tool functionality, improved tool reliability and maintainability, and improved documentation.

7. **Integrated Graphical User Interface (GUI)**

   Our goal is to provide an integrated development environment that enables the user to access the individual tools through a consistent windowed interface. Much of the user-visible work in Phase II will take advantage of features provided by the Eclipse Rich Client Platform development platform (see http://www.eclipse.org/rcp). Lower-level code improvements will take advantage of features provided by the Eclipse Java Development Tools (see http://www.eclipse.org/jdt). We also will provide facilities for user-defined visualization of automata specifications and simulated executions.

8. **User documentation**

   We will develop a comprehensive set of user manuals covering each individual tool developed within the project. Additionally we will develop user guides that introduce the users to the overall framework and its usage pattern, making it easier to use the tools developed in Phase II and any external tools integrated within the project, most notable PVS.